UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

TESIS DOCTORAL

Extensiones de bases de datos relacionales y deductivas: fundamentos teóricos e implementación

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Gabriel Aranda López

Directores

Susana Nieva Soto
Fernando Sáenz Pérez
Jaime Sánchez Hernández

Madrid, 2016

Extensiones de bases de datos relacionales y deductivas: fundamentos teóricos e implementación

TESIS DOCTORAL

Departamento de Sistemas Informáticos y Computación,
Facultad de Informática,
Universidad Complutense de Madrid

Autor:
Gabriel Aranda López

Directores:
Susana Nieva Soto
Fernando Sáenz Pérez
Jaime Sánchez Hernández

Tesis doctoral en formato publicaciones presentada por Gabriel Aranda López en el Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid para la obtención del título de doctor en Ingeniería Informática.

Terminada en Madrid el 20 de Octubre de 2015.

Nube de palabras

Índice general

Abstract

In this work we present some contributions to the field of database languages. We consider three general goals:

1. Improve the expressiveness of current database languages.
2. Develop formal semantics for our proposal of extended database languages.
3. Implement these semantics into practical database systems.

We have followed these steps moving in different database fields. On the one hand, in the deductive database field, we have proposed $HH_{\neg}(\mathcal{C})$ which extends deductive database languages allowing hypothetical queries and universal quantifications. On the other hand, we have moved to the relational database field and proposed *HR-SQL* that incorporates hypothetical queries as well as recursive definitions aimed to overcome some expressive limitations of standard database languages. Next, we introduce both proposals.

The scheme of Hereditary Harrop formulas with constraints, $HH(\mathcal{C})$, was proposed as a basis for Constraint Logic Programming languages. In the same way that Datalog emerges from logic programming as a deductive database language, such formulas can support a very expressive framework for constraint deductive databases, incorporating the intuitionistic implication that allows hypothetical queries and the use of quantifiers even in the constraint language. As negation is needed in the database field, $HH(\mathcal{C})$ is extended with negation to get $HH_{\neg}(\mathcal{C})$. The second chapter of this work presents the theoretical foundations of $HH_{\neg}(\mathcal{C})$ and an implementation that shows the viability and expressive power of the proposal. Moreover, the language is designed in a flexible way in order to support different constraint systems. The implementation includes several domains, and it also supports aggregates and strong integrity constraints as found in database languages. The formal semantics of the language is defined by a proof-theoretic calculus, and for the operational mechanism we use a stratified fixpoint semantics, which is proved to be sound and complete w.r.t. the former. Hypothetical queries and aggregates require a more involved stratification than the common one used in Datalog. The resulting fixpoint semantics constitutes a suitable foundation for the system implementation.

The Structured Query Language (SQL) is one of the most recognized and used database languages. It can be considered as a declarative programming language, but in its origin it lacked recursion. Although nowadays there are SQL database systems that partially support recursion, current database systems supporting recursive SQL impose restrictions on queries such as linearity, and do not allow mutual recursion. In addition, those systems are not founded on a formal semantics.

In the third chapter of this work we introduce the database language and prototype *R-SQL* that is an approach to overcome those drawbacks. Other useful aspect that has been studied

in the field of deductive databases is the use of hypothetical queries. We present a system, called *HR-SQL*, that enhances *R-SQL* in two main aspects.

On the one hand, it incorporates hypothetical queries as well as recursive and hypothetical view definitions, in a novel way which cannot be found in any other SQL system. In particular, allowing both positive and negative assumptions. All these features have been founded by extending the fixpoint semantics of *R-SQL*. On the other hand, the implementation of *HR-SQL* we have developed improves the efficiency of the previous prototype and is integrated in a commercial DBMS. We have also conducted some experiments to analyze its performance.

Keywords

Deductive Databases, Constraints, Hereditary Harrop Formulas, Fixpoint Semantics, Relational Databases, SQL, Recursion, Hypotheses.

UNESCO Categories

120312 Data banks

120323 Programming Languages

Resumen

En esta memoria hacemos contribuciones dentro del campo de los lenguajes de bases de datos. Nos hemos propuesto tres objetivos fundamentales:

1. Mejorar la expresividad de los lenguajes de bases de datos actuales.
2. Desarrollar semánticas formales para nuestras propuestas de lenguajes de bases de datos extendidos.
3. Llevar a cabo la implementación de las semánticas anteriores en sistemas de bases de datos prácticos.

Hemos conseguido estos tres objetivos en distintas áreas dentro de las bases de datos. Por un lado, en el campo de las bases de datos deductivas, proponemos $HH_{\neg}(\mathcal{C})$. Este lenguaje extiende las capacidades de los lenguajes de bases de datos deductivos con restricciones dado que permite consultas hipotéticas y cuantificación universal. Por otro lado, utilizamos el estudio dentro de las bases de datos deductivas y lo aplicamos a las bases de datos relacionales. En concreto proponemos *HR-SQL* que incorpora consultas hipotéticas y definiciones recursivas no lineales y mutuamente recursivas. La idea tras esta propuesta es superar algunas limitaciones expresivas del lenguaje estándar de definición de bases de datos SQL. A continuación introducimos ambas aproximaciones.
Las fórmulas de Harrop hereditarias con restricciones, $HH(\mathcal{C})$, se han usado como base para lenguajes de programación lógica con restricciones. Al igual que la programación lógica da soporte a lenguajes de bases de datos deductivas como Datalog (con restricciones), este marco se usa como base para un sistema de bases de datos deductivas que mejora la expresividad de los sistemas aparecidos hasta el momento.

En el segundo capítulo de esta memoria se muestran los resultados teóricos que fundamentan el lenguaje $HH_{\neg}(\mathcal{C})$ y una implementación concreta de este esquema que demuestra la viabilidad y expresividad del esquema. Las principales aportaciones con respecto a Datalog son la incorporación de la implicación intuicionista, que permite formular hipótesis, y el uso de cuantificadores incluso en el lenguaje de restricciones. El sistema está diseñado de forma que soporta diferentes sistemas de restricciones. La implementación incluye varios dominios concretos y también funciones de agregación y restricciones de integridad que son habituales en otros lenguajes de bases de datos relacionales. El significado del lenguaje se define mediante una semántica de pruebas y el mecanismo operacional se define mediante una semánica de punto fijo que es correcta y completa con respecto a la primera. Para el cómputo de las consultas hipotéticas y de las funciones de agregación se hace uso de una noción de estratificación más compleja que la que usa Datalog. La semántica de punto fijo desarrollada constituye un marco apropiado que lleva a la implementación de un sistema concreto.

El lenguaje de consultas estructurado SQL es el lenguaje estándar de definición y consulta de bases de datos relacionales. Se trata de un lenguaje declarativo que carecía de recursión

en sus orígenes. Sin embargo, hoy en día los lenguajes de bases de datos basados en SQL soportan la recursión de forma parcial imponiendo algunas restricciones como la linealidad de las definiciones recursivas y no permitiendo la recursión mutua. Además estas extensiones no están integradas en las semánticas disponibles para SLQ.

En el tercer capítulo de esta memoria proponemos el lenguaje y el sistema *R-SQL*. Esta aproximación supera las limitaciones de definiciones recursivas del estándar. Además hemos dotado al lenguaje inicial de un lenguaje de definición de vistas y de un lenguaje de consulta propios que permiten razonamiento hipotético, con lo que surge el lenguaje *HR-SQL*. Este segundo lenguaje mejora *R-SQL* en dos aspectos.

En primer lugar, incorpora hipótesis en vistas y consultas permitiendo razonamiento hipotético con suposiciones positivas y negativas. El fundamento semántico de *HR-SQL* está inspirado en la investigación para $HH_{\neg}(\mathcal{C})$. Por otro lado, se ha llevado a cabo una mejora de la eficiencia del cálculo del punto fijo en el sistema que se presenta como una capa superior sobre los sistemas de bases de datos relacionales existentes. Finalmente, presentamos los resultados de una comparativa de la eficiencia del sistema con otros sistemas de bases de datos actuales.

Palabras clave

Bases de datos deductivas, restricciones, fórmulas hereditarias de Harrop, semántica de punto fijo, bases de datos relacionales, SQL, recursión, hipótesis.

Códigos UNESCO

120312 Bancos de datos

120323 Lenguajes de Programación

Agradecimientos

Quisiera agradecer a Francisco Javier López Fraguas por darme la oportunidad de trabajar en el Grupo de Programación Declarativa y dedicarme a la investigación durante los primeros años de mi vida profesional.

Gracias también a mis directores Susana, Fernando y Jaime por introducirme en el mundo de la investigación y por la dedicación mostrada en sus explicaciones y revisiones que han permitido realizar este trabajo.

La presente tesis se enmarca dentro del trabajo desarrollado en el Grupo de Programación Declarativa de la Universidad Complutense de Madrid (grupo 910502 del catálogo de grupos reconocidos por la UCM) y ha contado con el apoyo de los siguientes proyectos de investigación:

- **PROMETIDOS-CM**. Programa Métodos Rigurosos de Desarrollo de Software de la Comunidad de Madrid (S-2009/TIC-1465).
- **FAST-STAMP**. Proyecto Foundations and Applications of declarative Software Technologies del Ministerio de Ciencia e Innovación (TIN2008-06622-C03).
- **CAVI-ART**. Proyecto Computer Assisted ValIdation by Analysis, annotation, pRoof, and Testing del Ministerio de Economía y Competitividad (TIN2013-44742-C4-3-R).
- **NGREENS** Proyecto Next-Generation Energy-Efficient Secure Software Software-CM de la Comunidad de Madrid (S2013/ICE-2731).

Además se ha contado con las ayudas al grupo de investigación mediante las convocatorias de referencias UCM-BSCH-GR35/10-A-910502 y UCM-BSCH-GR3/14-910502.

Parte I

Memoria

Capítulo 1

Introducción

Las bases de datos son una componente esencial de cualquier negocio o actividad empresarial relacionada con banca, enseñanza, telecomunicaciones o comercio, entre otros ejemplos [110, 122]. Están presentes habitualmente en nuestra actividad cotidiana: cuando navegamos por la red o hacemos compras online se está accediendo o actualizando una base de datos aunque no siempre seamos conscientes de ello.

Para gestionar una gran cantidad de información, un sistema gestor de bases de datos relacionales (en adelante SGBDR) debe proporcionar al usuario dos herramientas fundamentales. En primer lugar una estructura para almacenar los datos de forma ordenada. En segundo lugar un mecanismo de consulta y manipulación de datos sencillo y eficiente. En esta tesis estudiamos diferentes tipos de bases de datos: *bases de datos relacionales* (en adelante BDR), las *bases de de datos deductivas* (en adelante BDD) y, dentro de las segundas, nos centraremos en las *bases de datos con restricciones*. En este capítulo introducimos algunas ideas generales de las mismas.

Los SGBDR han sido objeto de estudio durante más de cuatro décadas [28, 92, 92, 36]. El SGBDR debe proporcionar al usuario un lenguaje de base de datos que permita definir la información extensionalmente en forma de tablas e intensionalmente en forma de vistas. Además debe proporcionar un lenguaje de consulta que permita acceder a una base de datos para recuperar información. Dada la gran cantidad de información que contienen las bases de datos actuales (como la de un banco por ejemplo) es importante diseñar lenguajes que se encarguen de esta tarea de forma eficiente. También es importante tratar de aportar la mayor expresividad posible a estos lenguajes para permitir al usuario recuperar información de forma sintética.

Los fundamentos de las BDR los encontramos en el modelo relacional que incluye como lenguajes formales el *álgebra relacional* [28], el *cálculo relacional de tuplas* [26, 25] y el *cálculo relacional de dominios* [27]. El modelo relacional es el más utilizado en la actualidad para implementar bases de datos. El álgebra relacional (en adelante AR) fundamenta el lenguaje estructurado de consultas (en adelante SQL por sus siglas en inglés) que es reconocido como lenguaje de bases de datos estándar por el Instituto Nacional estadounidense de estándares (en adelante ANSI por sus siglas en inglés) y también por la Organización Internacional para la Estandarización (en adelante OSI por sus siglas en inglés) [36]. Sin embargo, este modelo se ha mostrado insuficiente en la formulación de consultas. Un defecto importante es el uso limitado de recursión dado que no permite expresar consultas como el cierre transitivo de un grafo. Este tipo de consultas puede expresarse en la lógica de predicados y en sistemas de BDD [102]. En la actualidad la mayoría de los SGBDR que utilizan SQL no se ajustan al

estándar dado que permiten duplicados por ejemplo. Sin embargo, sí lo hacen en cuanto al tratamiento de la recursión restringiéndola al caso lineal y no permitiendo recursión mutua. Tan solo algunos sistemas en el entorno académico que manejan SQL [68, 103] permiten un uso más general de la recursión.

La aplicación de la programación lógica (en adelante PL) al campo de las bases de datos da lugar a las BDD [89, 11, 62, 92, 1]. Una base de datos deductiva incluye mecanismos para definir reglas que pueden deducir información adicional a partir de unos hechos almacenados. Las reglas se especifican mediante un lenguaje declarativo y posteriormente haciendo uso de un motor de inferencia se deduce nueva información. La mayoría de los sistemas de BDD utilizan el lenguaje Datalog [103] que surge como una extensión de Prolog para bases de bases de datos y que sigue siendo un referente en este campo [11, 126, 39, 20, 99, 100]. Datalog utiliza técnicas de estratificación para incorporar negación y recursión en sus bases de datos. Las BDD se aplican en diferentes áreas de la ciencia como la educación y la inteligencia artificial. Podemos encontrar un gran número de sistemas de BDD como XSB [104], bddbddb [65], LDL++ [3], DES [103], ConceptBase [54], QL [90], DLV [68], LogiQL [41] y 4QL [71].

La investigación en bases de datos con restricciones [64, 95] comenzó con el objetivo de extender la expresividad de las BDD al igual que la programación lógica con restricciones (en adelante CLP por sus siglas en inglés) extiende a la PL [52]. En este campo se avanzó sobre todo centrándose en los lenguajes de consulta sin recursión y con restricciones. Estos lenguajes llevaron a la investigación de problemas interesantes y derivaron en aplicaciones que se usan en muchas áreas como representación de la información espacial [46], la representación de datos espacio-temporales [105] y la bioinformática [94].

El esquema $HH(\mathcal{C})$ [66] se propuso originalmente como un lenguaje de programación lógica extendido con restricciones y cuantificadores. Este lenguaje está basado en la lógica intuicionista [72] y utiliza las fórmulas de Harrop hereditarias (HH) que fundamentan λ-Prolog [74] junto con restricciones que pertenecen a un sistema de restricciones $\mathcal{C}$ que parametriza el esquema. $HH(\mathcal{C})$ mejora la expresividad de CLP dado que permite objetivos que incluyen disyunciones, implicaciones, y cuantificadores universales y existenciales.

Situándonos en el contexto de los lenguajes de bases de datos, el objetivo fundamental de esta tesis es añadir, de forma bien fundamentada, expresividad a los lenguajes de consulta de bases de datos deductivas y relacionales. Otro de los objetivos de esta tesis es trasladar los fundamentos semánticos estudiados a la implementación de sistemas de bases de datos concretos. En particular, presentamos los trabajos llevados a cabo para implementar dos sistemas: uno deductivo, basado en el lenguaje $HH_{\neg}(\mathcal{C})$ (Hereditary Harrop formulas with Negation and Constraints) y uno relacional, basado en el lenguaje *HR-SQL* (Hypothetical and Recursive Structured Query Language) que aparece por primera vez en las publicaciones asociadas a esta tesis.

$HH_{\neg}(\mathcal{C})$ [83] surge con la idea de adaptar $HH(\mathcal{C})$ como lenguaje de bases datos, para lo que es necesario incorporar la negación al lenguaje para dar soporte a operaciones entre conjuntos como la diferencia y conseguir así completitud con respecto al AR. Lo novedoso de esta aproximación es la aplicación de los elementos expresivos que provienen de la lógica HH a las bases de datos. La implicación anidada permite formular consultas hipotéticas y constituye una de las principales aportaciones de esta tesis. Además se incorporan al lenguaje de base de datos otras funcionalidades de la lógica $HH(\mathcal{C})$ como son los cuantificadores existenciales y universales, y las restricciones. Otra de las aportaciones que presentamos en esta memoria es la incorporación de funciones de agregación y restricciones de integridad a $HH_{\neg}(\mathcal{C})$. Se pueden encontrar diferentes propuestas de trabajos sobre incorporación de funciones de agregación

tanto a las bases de datos con restricciones geométricas (véase el capítulo 6 de [64]) como a las BDD [31, 91, 130, 131]. Por su parte, las restricciones de consistencia de los datos son también conocidas como *restricciones fuertes de integridad* en el contexto de las BDD [20, 63, 55], y no se deben confundir con las restricciones del sistema de $HH_\neg(\mathcal{C})$ pertenecientes al sistema de restricciones $\mathcal{C}$ que parametriza el esquema. Las restricciones de integridad garantizan un uso seguro de la base de datos y son, por ejemplo, la clave primaria y la clave ajena. En $HH_\neg(\mathcal{C})$ el cálculo de funciones de agregación se delega en los resolutores del sistema de restricciones y las restricciones de integridad se calculan también utilizando los resolutores que, en este caso, devuelven *cierto* o *falso* según se cumpla o no una restricción concreta.

De la idea de trasladar las funcionalidades y formalismos de $HH_\neg(\mathcal{C})$ a un lenguaje relacional surge *HR-SQL*. El lenguaje, sus fundamentos semánticos y el sistema que los implementa son un resultado de esta tesis, que se ha abordado de manera incremental. En primer lugar, con el objetivo de trasladar la expresividad de las definiciones recursivas a las bases de datos relacionales, se desarrolló el lenguaje *R-SQL* que permite definiciones recursivas de relaciones no lineales y mutuamente recursivas, superando así la limitación de recursión del estándar SQL-99 [36]. Los fundamentos semánticos de *R-SQL* son próximos a los de $HH_\neg(\mathcal{C})$ y están también basados en una semántica de punto fijo estratificada que calcula el significado de una base de datos por capas o estratos.

La siguiente funcionalidad de $HH_\neg(\mathcal{C})$ que trasladamos al marco relacional es la capacidad de plantear hipótesis en vistas y consultas. Usando de nuevo una semántica similar a la de $HH_\neg(\mathcal{C})$ ampliamos los fundamentos de *R-SQL* para desarrollar *HR-SQL*. De esta forma se proporciona significado a un lenguaje muy cercano a SQL que permite recursión extendida y razonamiento hipotético. El sistema que implementa *HR-SQL* hace uso de los sistemas de bases de datos relacionales existentes y los extiende con capacidades habituales de los lenguajes de bases de datos deductivas (como la posibilidad definir relaciones recursivas no lineales o mutuamente recursivas) y otras capacidades que provienen de $HH_\neg(\mathcal{C})$ (como el manejo de hipótesis en vistas y consultas).

1.1. Motivación

Para motivar el trabajo y con el objetivo de mostrar las ventajas y expresividad de los marcos propuestos, mostramos algunos ejemplos de bases de datos y consultas con $HH_\neg(\mathcal{C})$ y *HR-SQL*.

En el contexto de las bases de datos deductivas utilizamos el término predicado que se corresponde con el término relación en bases de datos relaciones. De igual forma decimos que los objetivos se corresponden con las consultas. Además podemos distinguir dos componentes en una base de datos: la base de de datos *extensional* que se compone de predicados que están definidos mediante hechos y la base de datos *intensional* que se define mediante predicados que contienen al menos una regla. A continuación presentamos una colección de ejemplos que aceptan los sistemas $HH_\neg(\mathcal{C})$ y *HR-SQL* en cada caso.

El lenguaje $HH_\neg(\mathcal{C})$

En el primer ejemplo que presentamos de $HH_\neg(\mathcal{C})$ definimos una base de datos para las asignaturas cursadas por alumnos. La sintaxis concreta del lenguaje que se usa es muy cercana a Prolog. Además se usa `not(A)` para representar la negación de un átomo `A` y el símbolo

=> para las implicaciones anidadas (en una consulta o en el cuerpo de una regla o cláusula). Utilizamos también % para introducir comentarios en el código.

Hemos implementado varios sistemas de restricciones para $HH_{\neg}(\mathcal{C})$: booleanos, reales, enteros y dominios finitos. Cada sistema de restricciones tiene asociado su dominio correspondiente que es necesario definir explícitamente en el caso de un dominio finito. En el ejemplo, utilizamos el dominio de los números reales y dos dominios enumerados que definimos explícitamente: uno para los nombres de alumnos (**alum_dt**) y otro para las asignaturas (**asig_dt**). Dichos dominios se definen en $HH_{\neg}(\mathcal{C})$ como:

```
domain(alum_dt,[angela, david, joseluis, nicolas]).
domain(asig_dt,[introduccion_programacion,
                programacion_declarativa,
                programacion_funcional,
                programacion_logica]).
```

donde **domain** es la palabra reservada del sistema para definir un nuevo dominio finito y encontramos entre corchetes los valores respectivos de estos dominios.

La relación que definimos a continuación proporciona información sobre el nombre del **Alumno**, la **Asignatura** cursada y la **Nota** obtenida.

```
% curso(Alumno, Asignatura, Nota)
curso(angela,  introduccion_programacion, 5.0).
curso(nicolas, introduccion_programacion, 7.0).
curso(david,   introduccion_programacion, 2.0).
curso(angela,  programacion_declarativa,  3.0).
```

En $HH_{\neg}(\mathcal{C})$ también es necesario hacer declaración explícita de los tipos de los predicados. Al igual que sucede con las BDR la noción de tipo está estrechamente ligada a los dominios denotados por los sistemas de restricciones correspondientes. Para la relación **curso** introducimos su declaración de tipo:

```
type(curso(alum_dt,asig_dt,real)).
```

Continuamos con la definición de la base de datos introduciendo un predicado que determina que para poder matricularse de la asignatura **programacion_declarativa_avanzada** es necesario haber aprobado (y cursado) **introduccion_programacion** y haber cursado **programacion_declarativa** (aunque no necesariamente haber aprobado esta segunda):

```
% matriculaPDA(Alumno, Asignatura).
matriculaPDA(Alumno, programacion_declarativa_avanzada):-
  curso(Alumno, introduccion_programacion, Nota),
  Nota>=5.0,
  curso(Alum, programacion_declarativa, X).
```

A continuación presentamos algunos ejemplos de consultas a esta base de datos. Dado que el lenguaje incorpora negación, una consulta puede determinar quién no puede matricularse en **programacion_declarativa_avanzada**:

```
hhnc> not(matriculaPDA(Alumno, programacion_declarativa_avanzada)).
```

La respuesta en nuestro sistema de bases de datos es *una restricción*:

```
Alumno/= angela
```

que especifica que cualquier alumno distinto de Angela no puede matricularse, dado que ella es la única que ha aprobado y cursado las asignaturas requeridas.

Además de la posibilidad de plantear consultas hipotéticas, una de las aplicaciones que presenta este trabajo es el uso de funciones de agregación. Un ejemplo de la combinación de ambas es la siguiente consulta: suponiendo que el alumno José Luis obtuviese un 9.0 en la asignatura **introduccion_programacion** ¿cuál sería la media de las calificaciones de los alumnos de esta asignatura?

```
hhnc> curso(joseluis, introduccion_programacion, 9.0)=>
      Avg=avg(curso(Alumno, introduccion_programacion, Nota), Nota).
```

Las funciones de agregación se presentan dentro de las restricciones del lenguaje y se resuelven enviándolas al resoluto correspondiente de $HH_{\neg}(\mathcal{C})$. En este caso la función media (**avg**) tiene dos argumentos: el predicado al que se aplica (**curso**) y la variable sobre la que se calcula la media (**Nota**), y se resuelve en el sistema de restricciones reales. La respuesta a la consulta formulada es la siguiente restricción:

```
Avg = 5.75.
```

Para resolver una restricción de integridad se genera una restricción de nuestro sistema de restricciones que se envía a los resolutores utilizados. En el ejemplo, para especificar que que el par (**Alumno**, **Asignatura**) conforma la clave primaria del predicado **curso** se utiliza la siguiente declaración al definir la base de datos:

```
:- pk(curso(Alumno, Asignatura, Nota),(Alumno, Asignatura)).
```

Otra de las ventajas de trabajar con este lenguaje es la posibilidad de tratar con ciclos dentro de un grafo. Supongamos que añadimos una nueva definición de predicado para especificar de forma más sencilla cuándo una asignatura es prerrequisito de otra siguiendo la formulación que aparece en [102]. En la siguiente relación encontramos una parte extensional definida mediante dos hechos, y otra intensional, definida mediante una regla:

```
% pre(Asignatura, Asignatura).
pre(programacion_funcional, introduccion_programacion).
pre(programacion_logica, programacion_funcional).

pre(Pre, Post) :- pre(Pre, Asignatura),
                  pre(Asignatura, Post).
```

Podemos preguntar si al añadir un determinado prerrequisito se introduce un ciclo:

```
hhnc> pre(introduccion_programacion, programacion_logica)=>pre(X, X).
```

La respuesta es *cierto*.

A continuación presentamos cómo expresar también este ejemplo en el segundo lenguaje de bases de datos presentado en esta tesis: *HR-SQL*.

El lenguaje *HR-SQL*

En *HR-SQL* se definen relaciones asignando instrucciones *select* (del lenguaje de consulta SQL) a nombres de relación junto con su esquema (un esquema se compone de variables junto con sus tipos correspondientes que provienen del estándar de SQL). En este lenguaje distinguimos también la parte extensional de la intensional en una base de datos.

Comenzamos definiendo la relación **curso** que establece la correspondencia entre cada **alumno** y la **nota** obtenida en una **asignatura** determinada (al igual que el predicado de idéntico nombre introducido previamente). Para definir la parte extensional de una base de datos en *HR-SQL* utilizamos instrucciones *from-less* (siguiendo la nomenclatura inglesa) que permiten algunos SGBDR para definir tuplas sin origen de datos implícito sino explícito[1].

```
curso (alumno varchar(20), asignatura varchar(20), nota float) :=
 select 'Angela',  'Introduccion programacion', 5.0 union
 select 'Nicolas', 'Introduccion programacion', 7.0 union
 select 'David',   'Introduccion programacion', 2.0 union
 select 'Angela',  'Programacion declarativa',  3.0;
```

La relación que determina quién puede matricularse en **Programacion declarativa avanzada** se puede definir haciendo uso de dos relaciones auxiliares **aprobarIP** y **cursarPD**:

```
aprobarIP(alumno varchar(20)):=
  select curso.alumno from curso where
   curso.asignatura = 'Introduccion programación' and
   curso.nota>=5.0;

cursarPD(alumno varchar(20)):=
  select curso.alumno from curso where
   curso.asignatura = 'Programacion declarativa';

matriculaPDA(alumno varchar(20)):=
  select aprobarIP.alumno from aprobarIP,cursarPD where
   aprobarIP.alumno = cursarPD.alumno;
```

La consulta de quién no puede matricularse en **Programacion declarativa avanzada** se formula como:

```
hr-sql> select curso.alumno from curso
        except
        select * from matriculaPDA;
```

En lugar de restricciones, *HR-SQL* devuelve tuplas con los valores correspondientes como resultado:

$$[(\texttt{David},)(\texttt{Nicolas},)]$$

El resultado se presenta en forma de tuplas unitarias siguiendo la formulación que devuelve el sistema implementado que se incorpora en un SGBDR. En este caso, el sistema gestor es PostgreSQL y *HR-SQL* utiliza su notación al devolver la respuesta a una consulta.

[1] La tabla *dual* de Oracle consigue un efecto similar para devolver constantes o, en general, resultados de calcular expresiones.

Con el siguiente ejemplo presentamos una consulta hipotética equivalente a la introducida en $HH_{\neg}(C)$ que utiliza la función de agregación **avg**. Para incluir hipótesis en una consulta, *HR-SQL* incorpora la construcción **assume** <*Hipótesis*> **in** previa a la consulta de SQL que devuelve el resultado. La consulta se formula por tanto como:

```
hr-sql> assume  'Joseluis', 'Introduccion Programacion', 9.0
        in cursoIP
        select avg(nota) from cursoIP;
```

donde **cursoIP** es una relación auxiliar que devuelve los alumnos matriculados en **Introduccion Programacion**. La respuesta es la tupla unitaria con el valor de la función de agregación aplicada a la relación **curso** teniendo en cuenta la hipótesis incorporada: **[(5.75,)]**.

La formulación del predicado **pre** se especifica en *HR-SQL* de la siguiente forma[2]:

```
pre(pred varchar(20), pos varchar(20)) :=
  select 'Programacion funcional', 'Introduccion programacion'
  union
  select 'Programacion logica', 'Programacion funcional'
  union
  select pre1.pred, pre2.pos from pre as pre1,pre as pre2
   where pre1.pos = pre2.pred;
```

Definimos a continuación la consulta equivalente para obtener qué asignaturas forman parte de un ciclo cuando se asume una nueva tupla en el grafo de prerequisitos:

```
hr-sql> assume select 'Introduccion programacion',
                      'Programacion logica' in pre
        select pre.pred from pre where pre.pred = pre.pos;
```

Como hemos señalado, además del razonamiento hipotético, *HR-SQL* extiende la recursión de SQL-99. Por ejemplo, con nuestro lenguaje podemos definir de manera sencilla dos relaciones mutuamente recursivas para representar respectivamente los números pares e impares hasta 100:

```
par(x integer) :=
  select 0 union
  select impar.x+1 from impar;

impar(x integer) :=
  select par.x+1 from par where par.x<100;
```

Finalmente, como ejemplo relación no lineal proponemos la siguiente representación para los números de Fibonacci inferiores también a 100:

```
fib(n integer, f integer) :=
  select 0,1 union
  select 1,1 union
  select fib1.n+1,fib1.f+fib2.f from fib as fib1, fib as fib2
    where fib1.n=fib2.n+1 and fib1.n<100.
```

[2]La implementación actual de *HR-SQL* permite introducir *alias* en la sintaxis del lenguaje, si bien esta característica no aparece reflejada en las publicaciones asociadas a esta tesis.

Como hemos señalado, el estándar SQL-99 no admite este tipo de definiciones mutuamente recursivas. Dado que las primitivas aritméticas pueden introducir relaciones infinitas (y no terminación en el cómputo) utilizamos la condición que aparece en la cláusula **where** (`fib1.n<100`) para limitar el número de llamadas.

Una vez motivadas algunas de las capacidades expresivas que proporcionan estos lenguajes en los contextos deductivo y relacional pasamos a presentar los objetivos y aportaciones de esta tesis.

1.2. Objetivos y aportaciones: de $HH_{\neg}(\mathcal{C})$ a *HR-SQL*

Comenzamos este trabajo con el objetivo de extender los trabajos teóricos [83] para fundamentar e implementar $HH_{\neg}(\mathcal{C})$ como lenguaje de consulta de bases de datos deductivas. Además nos propusimos incorporar funciones de agregación y restricciones de integridad. A lo largo del desarrollo de la tesis se ha trabajado en estos objetivos.

En concreto, en [A.1] se presenta la primera implementación de $HH_{\neg}(\mathcal{C})$. En [A.3] abordamos la incorporación de las funciones de agregación a $HH_{\neg}(\mathcal{C})$ aprovechando la semántica estratificada de punto fijo para el cálculo de agregados. De forma similar, en [A.2] incorporamos las restricciones de integridad para las bases de datos deductivas utilizando el lenguaje del sistema de restricciones para especificarlas de forma sencilla.

Nuestra investigación se centra asimismo en trasladar ciertas ventajas de las bases de datos deductivas a un SGBDR y también utilizar técnicas semánticas propias de las BDD para formalizar el modelo relacional. En concreto, se trata de incorporar a los sistemas de bases de datos relacionales actuales un modelo de recursión más expresivo que permita la recursión no lineal y la recursión mutua así como el manejo de hipótesis en vistas y consultas. Con esta idea surge *HR-SQL* como un sistema que utiliza un lenguaje de base relacional que usa técnicas deductivas para el cómputo de punto fijo.

En [B.1] definimos el lenguaje *R-SQL*. En esta publicación presentamos una semántica de punto fijo estratificada que proporciona significado a bases de datos del lenguaje que permite definiciones recursivas no lineales y mutuamente recursivas. Además proponemos la primera implementación del sistema. Después, en el artículo [B.2] presentamos *HR-SQL* que permite incorporar hipótesis en vistas y consultas. Como veremos al final de la sección 1.5 existen otros trabajos sobre razonamiento hipotético en los SGBDR. Sin embargo, *HR-SQL* extiende la expresividad a la hora de incorporar hipótesis en vistas y consultas dado que permite hacer suposiciones positivas y negativas.

Respecto a los fundamentos teóricos que sustentan este trabajo presentamos una semántica de punto fijo estratificada para $HH_{\neg}(\mathcal{C})$ (que sirve de semántica operacional para la implementación de un sistema concreto). También se presenta una semántica de pruebas desarrollada previamente [83] y se demuestra que la semántica de punto fijo es correcta y completa con respecto a ella. El lenguaje $HH_{\neg}(\mathcal{C})$ es paramétrico con respecto al sistema genérico de restricciones $\mathcal{C}$, que al ser sustituido por un determinado sistema de restricciones da lugar a una instancia concreta. Presentamos también una semántica de punto fijo para el lenguaje *HR-SQL* inspirada en la semántica de punto fijo que fundamenta $HH_{\neg}(\mathcal{C})$. De esta forma proporcionamos significado a las bases de datos del lenguaje así como a las consultas y vistas que pueden contener hipótesis.

Respecto a la implementación presentamos los sistemas $HH_{\neg}(\mathcal{C})$ y *HR-SQL*. Al igual que ocurre con la definición semántica, hemos implementado el sistema $HH_{\neg}(\mathcal{C})$ de forma independiente del sistema de restricciones concreto. Asimismo, hemos implementado varios

sistemas de restricciones para el sistema: booleanos, dominios finitos y reales. Para la implementación de $HH_\neg(\mathcal{C})$ hemos usado el lenguaje SWI-Prolog [129]. Este sistema deductivo acepta como entrada las bases de datos del lenguaje y, haciendo uso del las restricciones asociadas al resolutor correspondiente, proporciona significado a sus predicados y consultas que combinan implicaciones, cuantificadores y restricciones. Además se han implementado las funciones de agregación delegando su cómputo al resolutor correspondiente del sistema. Así como también las restricciones de integridad haciendo uso de las restricciones del sistema para su implementación.

El sistema *HR-SQL* está implementado también siguiendo su formalización semántica y utilizando SWI-Prolog. Este sistema se ha diseñado como una capa sobre los sistemas relacionales comerciales. En particular puede trabajar con dos SGBDR: con PostgreSQL y con IBM DB2. La capa superior implementada en Prolog calcula la semántica de las bases de datos de *HR-SQL* y materializa las tablas resultantes en el SGBDR subyacente.

Utilizamos el lenguaje Python y el lenguaje de cuarta generación SQL PL (integrado en DB2 de IBM) como lenguajes intermedios que sirven para comunicar el sistema implementado en Prolog y el SGBDR. Nuestros sistemas crean programas en estos lenguajes intermedios y estos programas generan las bases de datos *HR-SQL* mediante instrucciones SQL estándar embebidas en ellos.

Hemos trabajado también en la eficiencia de *HR-SQL* con respecto a *R-SQL*. La noción de estratificación ha evolucionado desde una que aglutina el máximo número de relaciones en un mismo estrato, tanto en el caso $HH_\neg(\mathcal{C})$ como en la primera versión de *R-SQL*, a una que minimiza el número de relaciones en cada uno de los estratos para *HR-SQL*. Al minimizar las relaciones en los estratos se mejora la eficiencia dado que se reduce el número de iteraciones de los bucles utilizados para alcanzar el punto fijo. También el cómputo de punto fijo ha mejorado. En *HR-SQL* se separan las definiciones de relaciones recursivas en el caso base y el caso recursivo, extrayendo el primero del cuerpo del bucle en cada estrato. Así se evita recalcular inútilmente la parte no recursiva en las sucesivas iteraciones necesarias para alcanzar el punto fijo. Esta técnica es habitual en el campo de las BDD [122]. Finalmente usamos tablas temporales para obtener las tuplas que se añaden (o eliminan) a las relaciones de la base de datos al calcular vistas y consultas hipotéticas. Dado que las tablas temporales no generan entradas de *log* en el SGBDR ni demandan gestión de concurrencia han resultado una herramienta adecuada para nuestro fin, sin que ello conlleve una gran pérdida de rendimiento, si bien es cierto que esta funcionalidad no está disponible en todos los SGBDR actuales (véase la sección 3.6).

De este modo contribuimos en dos áreas en las bases de datos: por un lado, aportando y fundamentando el lenguaje $HH_\neg(\mathcal{C})$ en el área de las BDD, que extiende las capacidades de Datalog; y por otro lado, en el de las BDR, mejorando los SGBDR existentes al permitir hipótesis en vistas y consultas, así como un tratamiento más general de la recursión. También presentamos técnicas semánticas que son novedosas en ambos campos.

1.3. Organización del trabajo

La memoria se divide en cuatro capítulos (incluyendo este primer capítulo introductorio) con el siguiente contenido:

- En el capítulo 2 presentamos el lenguaje $HH_\neg(\mathcal{C})$, su sintaxis, la definición de su sistema de restricciones y las semánticas que fundamentan este lenguaje. También presentamos

el sistema implementado siguiendo la semántica de punto fijo. Con ello se resumen los contenidos de las publicaciones [A.1, A.2, A.3].

- En el capítulo 3 presentamos el marco teórico *HR-SQL*: su sintaxis y su semántica de punto fijo. Además presentamos la implementación de un sistema basado en este marco. En este capítulo se resumen las publicaciones [B.1, B.2, B.3].
- Finalmente en el capítulo 4 presentamos las conclusiones y planteamos líneas de trabajo futuro.

1.4. Publicaciones asociadas a la tesis

[A.1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez.
Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas.
En Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09), páginas 117–128. ACM Press, 2009.
→ Página 116

[A.2] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Incorporating Integrity Constraints to a Deductive Database System.
En XI Jornadas sobre Programación y Lenguajes, PROLE2011 (SISTEDES)
editores: Purificación Arenas, Victor M. Gulías y Pablo Nogueira, páginas 141–152, Septiembre, 2011.
→ Página 128

[A.3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
An Extended Constraint Deductive Database: Theory and implementation.
The Journal of Logic and Algebraic Programming, volumen 21, páginas 20–52, 2013.
→ Página 140

[B.1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Formalizing a Broader Recursion Coverage in SQL.
En Symposium on Practical Aspects of Declarative Languages (PADL'13), volumen 7752 de *LNCS*, páginas 93 – 108, 2013.
→ Página 176

[B.2] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Incorporating Hypothetical Views and Extended Recursion into SQL Database Systems.
En Ken Mcmillan, Aart Middeldorp, Geoff Sutcliffe, y Andrei Voronkov, editores, LPAR-19, volumen 26 de *EPiC Series*, páginas 9–22. EasyChair, 2014.
→ Página 192

[B.3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
R-SQL: An SQL Database System with Extended Recursion.
En Electronic Communications of the EASST, volumen 64: Programming and Computer Languages, páginas 1–18, 2013.
→ Página 206

A continuación presentamos el estado del arte.

1.5. Estado del arte

Comenzamos haciendo un repaso de modelos para las BDD y los sistemas a los que dan lugar. Hacemos énfasis en las diferentes aproximaciones para incorporar negación y agregación en los lenguajes de bases de datos deductivas dado que en la memoria presentamos cómo se incorporan ambas en $HH_{\neg}(\mathcal{C})$. A continuación se hace una revisión de distintos sistemas de bases de datos con restricciones y sus aplicaciones. Terminamos el capítulo presentando otras aproximaciones para la recursión y el razonamiento hipotético en las BDR.

1.5.1. Bases de datos deductivas

Los sistemas de BDD son aquellos que obtienen nuevos datos a través de un motor de inferencia. Puede incorporar gestor de transacciones, control de seguridad y control de persistencia, entre otras funcionalidades. Las BDD se llaman también bases de datos lógicas, dado que tienen su génesis en la PL. Una característica de las BDD, compartida con las bases de datos relacionales, es que sus lenguajes de consulta tienen la propiedad de ser *declarativos*. Esto significa que permite al usuario hacer una consulta planteando *qué* información quieren obtener, en vez de *cómo* realizar la operación.

Según [77] la incorporación de la lógica ha aportado un gran número de contribuciones a las bases de datos, entre las que se pueden destacar:

- Formalización de base de datos, consulta y respuesta a una consulta.
- El reconocimiento de que la programación lógica extiende a las bases de datos relacionales.
- Presentación de la semántica de múltiples clases de bases de datos que incluyen formas alternativas de negación y disyunción.
- Comprensión de las restricciones de integridad y la forma en que se pueden aprovechar al realizar actualizaciones y optimización de la semántica de consultas.
- Formalización y soluciones a los problemas de actualización de datos y de vistas.
- Comprensión de la recursión y la forma en que puede ser implementada prácticamente.
- Comprensión de las relaciones entre los sistemas basados en la lógica y los sistemas basados en el conocimiento [93].
- Formalización de la gestión de la información incompleta en sistemas de bases de conocimiento.
- Correspondencia entre formalismos alternativos de razonamiento no monótono y bases de datos y de conocimiento.

La mayoría de los sistemas deductivos están inspirados en Prolog. A la hora de diseñar e implementar se debe tener en cuenta:

- La estrategia de evaluación de Prolog puede conducir a cómputos infinitos debido a los predicados recursivos, incluso con programas sin negación o también en ausencia de símbolos de función o aritméticos. Sin embargo, en los lenguajes de BDR más extendidos basados en SQL se espera que las consultas terminen siempre.

- La corrección y completitud del método de evaluación.
- La cantidad de información es lo suficientemente grande como para formar parte del almacenamiento secundario en una aplicación típica de bases de datos. Para un buen rendimiento del sistema, el acceso eficiente a estos datos es crucial.
- Finalmente, un objetivo primordial de las bases de datos deductivas es tratar con un superconjunto del AR que permita recursión sin que ello conlleve un gran número de accesos a disco, que sea terminante.

El origen de las bases de datos deductivas se puede encontrar en trabajos relacionados con demostradores automáticos de problemas y en la PL. En otro estudio realizado también por Minker [76] se sugiere que Green y Raphael [40] fueron los primeros en relacionar la demostración de teoremas y la deducción en bases de datos. Estos desarrollaron una serie de sistemas consulta y respuesta que usaban una versión del principio de resolución de Robinson [98], demostrando así que la deducción se puede usar de manera sistemática en el contexto de las bases de datos.

Los primeros sistemas que implementan estas ideas son MRPPS [78], DEDUCE [22] y DADM [60]. El primero, MRPPS, era un intérprete que fue desarrollado por el grupo de Minker entre 1970 y 1978. De él se puede destacar que incluyó una de las primeras propuestas de consultas recursivas. DEDUCE fue implementado por IBM en la década de los 70, y usaba reglas basadas en cláusulas de Horn recursivas lineales por la izquierda. Finalmente en DADM se hizo explícita la diferencia entre la parte extensional e intensional de una base de datos y se presentaba la representación de la intensional en forma de *grafos de conexión*.

En 1976, van Emdem y Kowalski [32] mostraron que el mínimo punto fijo de un programa lógico con cláusulas de Horn era el modelo mínimo de Herbrand. Esto fundamentó la semántica de los programas lógicos, así como de las bases de datos deductivas, dado que la semántica operacional consiste en el cómputo de punto fijo asociado a una base de datos deductiva (al menos, a las basadas en evaluación ascendente).

Los primeros trabajos se centraron en establecer los objetivos de las BDD y el desarrollo de sus fundamentos semánticos. La siguiente fase se centró en el desarrollo de la evaluación eficiente de consultas. Henschen y Naqvi [49] propusieron una de las primeras técnicas eficientes para evaluar consultas en el contexto de las bases de datos. Tras esto, en un artículo de Ullman [121] se fijó un marco para la implementación. Con este fin, el autor se centró no solo en técnicas para evaluar consultas sino que también llamó la atención sobre el problema de la no terminación.

El área de las BDD alcanzó gran importancia en 1984 con el comienzo de tres importantes proyectos. El proyecto Nail! en Stanford, LDL en Austin y el proyecto de bases de datos deductivas ECRC representaron la mayor contribución a las bases de datos deductivas fuera de las universidades.

El proyecto ECRC fue coordinado por J. M.f Nicolas. La primera fase [17] llevó al estudio de los algoritmos de desarrollo de los primeros prototipos, comprobación de integridad y un sistema inicial que exploraba la comprobación de consistencia. La segunda fase trajo prototipos más funcionales: Megalog [12], DedGin [125], EKS-V1 [67]. El sistema EKS daba soporte a restricciones de integridad y algunas funciones de agregación que usaban recursión. De este proyecto se derivan investigaciones como las que llevó a cabo el Groupe Bull, que desarrolló bases de datos deductivas comerciales y orientadas a objetos.

El proyecto LDL [120] comenzó también en 1984. En 1986 se evidenció que la combinación de Prolog con las bases de datos relacionales no era una solución satisfactoria, así que

comenzaron con el desarrollo de técnicas ascendentes para el cálculo de la semántica de la base de datos. El prototipo LDL se desarrolló en 1988 y tuvo nuevas versiones entre 1989 y 1991. Este fue el primer sistema de bases deductivas de propósito general que estuvo disponible. Dicho sistema incorporaba negación estratificada y era compilado por un sistema que producía código C. Encontramos una presentación del lenguaje LDL en [80]. El sistema LDL++ en MCC [131] es el sucesor directo que comenzó en 1991. Este sistema incluye negación no estratificada y funciones de agregación. Actualmente el sistema LDL++ ha evolucionado al sistema Deals [109] (`http://wis.cs.ucla.edu/deals`).

El proyecto Nail! (*Not Another Implementation of Logic!*) comenzó en Stanford en 1985 siguiendo las ideas que aparecen en [121]. En colaboración con el grupo MCC apareció el primer artículo sobre conjuntos mágicos (*Magic Sets*) [8]. Se desarrolló un prototipo inicial [79] y finalmente fue abandonado dado que el paradigma puramente declarativo no resultaba cómodo para la realización de muchas aplicaciones.

El proyecto Aditi comenzó en 1988 en la Universidad de Melbourne. Las principales contribuciones de este proyecto son la formulación de una evaluación *naïve* que ha sido muy usada en trabajos posteriores [6], la adaptación de conjuntos mágicos para programas estratificados [5], indexación y optimización de programas con restricciones [61]. El trabajo del grupo se encaminó hacia el desarrollo de su prototipo, haciendo especial énfasis en las relaciones residentes en disco. Se puede ver una visión general de este sistema en [123].

El sistema ConceptBase [55] desarrollado en la Universidad de Passau y Aachen desde 1987 trataba de combinar reglas deductivas con un modelo de datos semántico. El sistema [56] también tiene soporte de restricciones de integridad. ConceptBase se ha usado en numerosas aplicaciones en universidades europeas y tiene una versión comercial.

El proyecto CORAL perteneciente a la Universidad de Wisconsin comenzó en 1988. La idea original era el desarrollo del algoritmo de plantillas mágicas (*Magic Templates*) [85], que ofrecía la posibilidad de usar tuplas no cerradas. Este proyecto aporta grandes contribuciones en el desarrollo de semánticas de multiconjuntos para PL y optimización cuando se trata de comprobaciones de duplicados [70]. Los primeros resultados de técnicas ascendentes eficientes con respecto a espacio aparecen en [82, 115]. Además la presentación de la evaluación de programas con funciones de agregación se puede encontrar en [113]. El resultado de que la evaluación ascendente domina asintóticamente a la descendente (en el contexto de programas con cláusulas de Horn) se obtuvo a través de este proyecto [114]. El primer prototipo del sistema CORAL estuvo operativo en 1990. Esta versión soportaba agregación no estratificada y negación, usando un algoritmo propuesto en [86]. Podemos encontrar una visión general del sistema en [87] y la implementación aparece descrita en [88]. La extensión que soporta características orientadas a objetos es Coral++ [111].

El proyecto XSB [104] es otro trabajo relacionado, fue coordinado por D.S. Warren. Se desarrolló un sistema que soporta negación estratificada y agregación (además de un meta-intérprete para programas bien fundamentados), tuplas no cerradas y relaciones residentes en disco. La implementación se basa en la resolución OLDT [116]. La máquina abstracta de Warren WAM (Warren Abstract Machine) [127], una máquina abstracta para implementar sistemas Prolog, se adaptó para usar la evaluación descendente que se usa en XSB.

El sistema educativo DES es un sistema de bases de datos deductivas desarrollado en la Universidad Complutense [103], es gratuito y de código abierto. Incluye lo lenguajes de programación Datalog, SQL y AR. Soporta negación estratificada, depuración declarativa, generación de casos de prueba para vistas SQL, funciones y predicados de agregación, y predicados *join*, restricciones de integridad fuertes, tablas memo [108] y consultas hipotéticas.

El interprete Inter4QL [71] está basado en un lenguaje de bases de datos llamado 4QL que permite negación en cuerpos y cabezas de las reglas. 4QL utiliza una semántica multivalorada de cuatro valores: *true, false, inconsistent* y *unknown*. Esta semántica proporciona significado para un tratamiento uniforme de lo que se denomina suposición del múndo cerrado (o CWA por sus siglas en inglés). Además el lenguaje puede representar otros formalismos como distintas variantes de razonamiento *por defecto*, razonamiento autoepistémico y otros formalismos para la desambiguación de información inconsistente.

El lenguaje de consulta lógico *LogiQL* [41] es un lenguaje de programación declarativo que proviene de Datalog y está desarrollado por *LogicBlox Inc.* para su motor de bases de datos *LogicBlox*. Se ha desarrollado utilizando técnicas eficientes para la evaluación de consultas, gestión de concurrencia, optimización del trabajo en red, análisis de programas así como para modelos de programación declarativos y reactivos.

El sistema bddbddb [65] (*BDD-Based Deductive DataBase* por sus siglas en inglés) es una implementación de Datalog que representa la información usando diagramas binarios de decisión. Estos diagramas son estructuras de datos que pueden representar de forma sintética relaciones con gran cantidad de datos y proporcionan un conjunto de operaciones muy eficiente. Esto hace que *bddbddb* pueda representar y operar con relaciones que contienen un número extremadamente grande de datos.

En [89] se encuentra una tabla comparativa de varios sistema que mostramos y ampliamos con sistemas actuales en la figura 1.1. Además hemos incluido también nuestro sistema $HH_{\neg}(\mathcal{C})$ en esta tabla. Los parámetros sobre los que hacemos comparativa son:

1. *Recursión* (Rec.). Muchos de los sistemas permiten usar recursión general. Sin embargo, algunos limitan la recursión a una serie de casos restringidos relacionados con búsqueda de grafos.

2. *Negación*. La mayoría de los sistemas permiten negación en el cuerpo de las reglas. Cuando esto ocurre suele haber más de un punto fijo mínimo y el sistema debe seleccionar uno de ellos en función del modelo pretendido.

3. *Agregación*. Un problema parecido al de la negación aparece con la agregación (suma, promedio, etc). Esto hace que aparezca más de un modelo mínimo que debemos discriminar.

Expresividad y modelos para la negación en bases de datos deductivas

Con respecto al fundamento semántico de otras propuestas para la incorporación de negación, destacamos:

- la aproximación de los *Modelos Estables* de Gelfond y Lifschitz [37]. Se trata de una semántica declarativa para programas con negación basada en lógica.

- La semántica bien fundada (*Well-Founded Semantics*) de Van Gelder, et al. [124]. En esta aproximación la idea principal es la de *unfounded set* que se usa para formalizar la negación.

Si compráramos la expresividad de nuestra propuesta con la de estos modelos para la negación, debemos señalar que las dos aproximaciones anteriores trabajan con instancias básicas (*ground* según la nomenclatura inglesa) en los cuerpos de las cláusulas. Por el contrario, los sistemas de restricciones de $HH_{\neg}(\mathcal{C})$ permiten representación intensional y respuestas más

Nombre	Desarrollado	Ref.	Rec.	Negación	Agregación
Aditi	U. Melbourne	[123]	Sí	Estratificada	Estratificada
bddbddb	U. Stanford	[65]	Sí	Estratificada	No
Concept Base	U. Aachen	[56]	Sí	Localmente Estratificada	No
CORAL	U. Wisconsin	[87]	Sí	Modularmete Estratificada	Modularmente Estratificada
DES	U. Complutense	[103]	Sí	Estratificada	Estratificada
EKS	ECRC	[67]	Sí	Estratificada	Estratificada
$HH_{\neg}(\mathcal{C})$	U. Complutense	[A.3]	Sí	Estratificada	Estratificada
Inter4QL	U. Varsovia	[71]	Sí	Semántica Multivalorada	No
LDL LDL++	MCC	[23]	Sí	Estratificada Restringida	Estratificada Restringida
LogicBlox	LogicBox Inc.	[41]	Sí	Semántica Multivalorada	Parcial
XSB	SUNY Stony Brook	[104]	Sí	Bien Fundada	Modularmente Estratificada

Figura 1.1: Comparativa de implementaciones de sistemas de bases de datos deductivas.

generales que si los limitamos a restricciones de igualdad de variables. Por ejemplo, las restricciones sobre reales permiten representar datos posiblemente infinitos. Sin embargo, no podemos representarlas con átomos básicos de forma directa. Además cabe señalar que tanto los modelos estables como la semántica bien fundada permiten bases de datos que no serían estratificables en nuestra aproximación. Es decir, el uso de la estratificación supone una limitación en cuanto a las bases de datos que es posible representar en el lenguaje. Sin embargo, dados los recursos de $HH_{\neg}(\mathcal{C})$ (el uso de restricciones, cuantifcadores e implicación) cuando se presenta una base de datos no estratificable en algunos casos se puede encontrar una base de datos de nuestro lenguaje que sea equivalente (veáse ejemplo 1 de [A.2]).

Finalizamos señalando que *answer set programming* [69] es otra aproximación de la programación declarativa que incorpora negación y es adecuada para trabajar con problemas de búsqueda de dificultad combinatoria. Está basada en *modelos estables* y utiliza resolutores como mecanismo computacional de inferencia. Esta propuesta incluye restricciones que se usan para resolver restricciones de integridad, para descartar modelos y obtener una respuesta. Sin embargo, en $HH_{\neg}(\mathcal{C})$ estas restricciones forman parte de la respuesta. Además esta aproximación no permite el manejo de implicación que hace nuestro sistema.

1.5.2. Bases de datos con restricciones

Una de las ventajas del uso de restricciones en el contexto de la PL es su capacidad para tratar con infinitos datos mediante el uso de representaciones finitas. Las bases de datos con restricciones [64] heredan esta característica.

La investigación en bases de datos con restricciones comenzó con el objetivo de definir una versión de CLP orientada a las bases de datos. El primer objetivo fue usar técnicas

ascendentes para procesar reglas Datalog usando además restricciones. Así se hacía posible el uso de CLP para aplicaciones de manera que los datos podían representarse como conjuntos de restricciones (por ejemplo, datos espaciales). Según avanzaba la investigación resultó que el problema de dar soporte a la recursión en presencia de restricciones no llegaba a ser resuelto de manera satisfactoria. Por tanto, en este campo se avanzó sobre todo centrándose en el caso no recursivo. Los lenguajes de consultas no recursivos con restricciones llevaron a la investigación de problemas interesantes y nada triviales y se usan en diferentes campos [29].

La idea de usar restricciones para representar objetivos se había discutido en el campo de las matemáticas (por Whitney [128] por ejemplo) y en el de la investigación operativa (por Dantzig [81]), y se había usado en algunas aplicaciones de gestión de bases de datos espaciales (la más notable CAD/CAM [106]).

En el campo de las bases de datos, Kanellatis et al [58] son los primeros en definir un marco de trabajo sistemático para el uso de restricciones como modelo de datos complejos y lenguajes de consulta sobre dichos datos.

Se han definido álgebras y cálculos para aplicaciones concretas siguiendo algunas líneas del modelo relacional. Estos sistemas se utilizan para dar modelos concretos a datos temporales, denominados eventos, y que aparecen en intervalos regulares. Este modelo aparece descrito en [57, 9] y también en el capítulo 13 de [64].

Otro modelo parecido es la propuesta de [46] para la representación de información del espacio en un GIS (*Geographic Information System*), como la intersección de semiplanos. El lenguaje de [46] es un caso particular del lenguaje de consulta con restricciones lineales y sin proyección. Otra propuesta en la misma dirección es la que aparece en [47], en la que se trata de incluir información de dependencia del dominio en la base de datos.

El sistema MLPQ (*Management of Linear Programming Queries*) es un sistema de bases de datos con restricciones lineales. Se desarrolló en la Universidad de Nebraska-Lincoln. La primera versión se presentó en [97] e incluye consultas SQL y programación lineal como funciones básicas para implementar agregados (como máximo y mínimo). La segunda versión se presentó en [59] e incluye consultas Datalog tanto recursivas como no recursivas y una interfaz gráfica de usuario con operadores espaciales de dos dimensiones. Se pueden encontrar dos grandes aplicaciones de MLPQ: la investigación operativa y el tratamiento con datos espaciales y espacio temporales.

El sistema DISCO (*Datalog with Integer Set of COnstraints*) es un sistema de bases de datos que implementa Datalog con restricciones booleanas sobre enteros o conjuntos de tipos enteros. Fue desarrollado en la Universidad de Nebraska. La primera versión del sistema fue presentada por Revesz en [18]. La segunda versión de DISCO está descrita en [105] e incluye restricciones de desigualdad e igualdad booleana sobre conjuntos de enteros. En general la igualdad y la desigualdad booleanas no pueden ser usadas de manera conjunta en una restricción.

El prototipo DEDALE es una de las primeras implementaciones de un sistema de bases de datos basado en un sistema de restricciones lineales. Es un proyecto de INRIA con el grupo VERSO y el grupo VERTIGO. El prototipo DEDALE se utiliza para aplicaciones geométricas en diversas áreas como GIS o bases de datos espacio temporales. El sistema se describe en [45] y su modelo de datos en [44].

Otras aplicaciones de las bases de datos con restricciones son:

- En visión por computador, para la indexación de bases de datos de imagenes en función de su forma y contorno [53, 43].
- En bioinformática, para el desarrollo de un autómata para la descodificación del genoma

humano, uno de los problemas más importantes en bioinfomática que se ha tratado usando el sistema LDL [94].

- También se usa en el modelado de entorno, para el desarrollo de mapas térmicos que se utilizan para evitar la propagación de fuego [48].

1.5.3. Bases de datos deductivas con razonamiento hipotético

Una de las principales aportaciones de este trabajo es la incorporación de razonamiento hipotético haciendo uso de implicaciones anidadas, una característica que es poco habitual en lenguajes de bases de datos de la que si encontramos trabajos en la PL (véase por ejemplo [72, 66, 4]).

Uno de los trabajos más importantes dentro del campo de las bases de datos deductivas es la contribución de Antony J. Bonner [13, 14]. La aproximación de Bonner consiste en una extensión de la lógica de cláusulas de Horn que permite consultas hipotéticas en el lenguaje, de una forma similar a nuestro enfoque. En los trabajos de Bonner se permite la adición o borrado de tuplas temporalmente en la base de datos. En ambos casos, incorporación (`A←B [add:C]`) y borrado (`A←B [del:C]`), el término atómico `C` se añade o borra cuando se realiza una consulta `A` a la base de datos extensional `B`. Su aproximación tiene una serie de limitaciones con respecto a $HH_{\neg}(\mathcal{C})$:

- $HH_{\neg}(\mathcal{C})$ permite cláusulas como antecedentes en la consultas hipotéticas, no solo un átomo, lo cual permite cambiar dinámicamente la base de datos intensional y no solo la extensional.

- El lenguaje de bases de datos que presentamos en este trabajo combina el razonamiento hipotético con restricciones y nuevas conectivas. El hecho de manejar todas estas características conjuntamente añade una expresividad a nuestra aproximación de la que adolece la propuesta de Bonner.

Un sistema reciente que implementa Datalog hipotético basado en técnicas de *tabling* es [102]. En este trabajo se implementa un cálculo de consultas hipotéticas que permite la aparición de átomos básicos tanto positivos `A`, como negativos `not(A)` en el cuerpo de una consulta.

- Si se formula la consulta sobre una relación por primera vez, se añade una nueva entrada a la *tabla de respuestas*. Se descompone la consulta, para ver si algún subconjunto de ella está presente en la tabla de respuestas y se elabora una sustitución lo más general posible que permanecerá en esta tabla para futuras consultas.

- Si se formula una consulta que está presente en la tabla de respuestas, el sistema responderá directamente devolviendo el valor de la tabla de forma inmediata.

- Para el caso de una consulta, o subconsulta, con un átomo negado, `not(A)`, se tratará de buscar una sustitución que satisfaga `A`. En caso de que no se encuentre, la consulta negativa puede ser probada.

Esta implementación se basa en la asunción del mundo cerrado de [122] e implementa las ideas de Bonner [15], mediante técnicas de *tabling* [108].

1.5.4. Bases de datos relacionales, uso de la recursión y el razonamiento hipotético

Una BDR es una base de datos que sigue el modelo relacional, el cual es el modelo más utilizado en la actualidad para implementar bases de datos. Permiten establecer relaciones entre los datos (guardados en tablas), y a través de dichas relaciones conectar los datos de las tablas correspondientes, de ahí proviene el nombre del modelo. Tras ser postuladas sus bases en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José (California), no tardó en consolidarse como un nuevo paradigma en los modelos de base de datos.

El estándar SQL-99 [36] incluye una sintaxis para la definición de vistas recursivas (véanse los capítulos 9 de [73], 4 de [110] y 10 de [36]). Para la formulación de una vista recursiva es necesario utilizar explícitamente las palabras reservadas `WITH RECURSIVE` evitando así que se creen vistas recursivas de forma accidental (aunque este no es un requisito que impongan todos los SGBDR) que pueden llevar a la no terminación del cómputo. Podemos formular este tipo de vistas temporales en muchos de los SGBDR actuales como son PostgreSQL, DB2 y Oracle. Por otro lado MySQL y Microsoft Access no permiten ningún tipo de definición recursiva.

Como señalamos en la primera subsección de este capítulo, existen una serie de limitaciones al de definir vistas recursivas en SQL [102].

- Se requiere el uso de `UNION ALL` a la hora de definir la unión del caso base y el caso recursivo para evitar el descarte de duplicados.
- Cuando se define el cierre transitivo de un grafo no se comprueba si las nuevas tuplas añadidas pertenecen a la definición recursiva, lo que lleva a la no terminación para algunos casos.
- No se permite más de una llamada a una misma relación en la definición recursiva,i.e., la recursión está limitada al caso lineal.
- No está permitida la recursión mutua.

En la literatura sobre semánticas de bases de datos SQL no encontramos una formalización que combine recursión y consultas hipotéticas de la forma en que se plantea en este trabajo. Sin embargo, hay algunos trabajos relacionados con el razonamiento hipotético BDR que introducimos a continuación. Estos trabajos pueden considerarse los primeros en abordar la inclusión de información hipotética en una base de datos relacional.

En [112] se permite una expresión limitada de consultas hipotéticas. Las suposiciones se calculan haciendo uso del operador de reemplazamiento, que mantiene la información supuesta hasta que la consulta termina. En el trabajo se abordan dos tipos de razonamiento hipotético con dos aproximaciones:

- Se permite añadir (`APPEND`), actualizar (`RETRIEVE`) y borrar (`DELETE`) un número (posiblemente 0) de tuplas en una base de datos.
- Se introduce el concepto de experto que permite especificar definiciones de relaciones en función de una decisión que se tomará más adelante.

Este trabajo no incluye la recursión y no se ha implementado en un sistema concreto.

En [42] se presenta el AR extendida para tratar con consultas hipotéticas (de la forma Q *when* $\{\{U\}\}$) haciendo uso de actualizaciones en la base de datos, pero sin recursión ni una

implementación concreta. El resultado de la consulta Q es el valor de la base de datos DB tras la ejecución de U.

La principal diferencia de estos trabajos con la aproximación *HR-SQL* es:

- *HR-SQL* permite un uso más general de la recursión permitiendo definiciones no lineales y mutuamente recursivas.
- Nuestra aproximación permite hacer suposiciones en consultas y vistas no solo de tuplas, sino también de relaciones intensionales de la base de datos.
- Nuestros desarrollos semánticos han servido de base para un sistema concreto que además se integra con los sistemas de bases de datos actuales como una capa adicional extendiendo el SGBDR.

Para concluir, destacamos de nuevo el sistema educativo DES [103] dado que soporta también SQL e hipótesis en consultas y vistas como *HR-SQL*. Este sistema admite también su misma sintaxis (incluyendo la palabra reservada `assume` para incorporar hipótesis). Actualmente funciona tanto en SWI-Prolog como en SICStus Prolog. Sin embargo, su funcionamiento es independiente del Prolog subyacente. SQL hipotético en DES se basó en principio en el trabajo de [42], i.e., se modificaban las relaciones afectadas por las hipótesis antes de una consulta y se restauraban las relaciones originales una vez que se devolvía el resultado. La implementación actual se fundamenta en el trabajo de Bonner [13] y su motor de inferencia deductivo traduce SQL a Datalog hipotético mediante técnicas de *tabling* [108]. Esta aproximación permite una expresividad similar a *HR-SQL* pero sus fundamentos semánticos son distintos y las implementaciones a las que dan lugar hacen uso también de técnicas distintas. En concreto *HR-SQL* implementa una semántica de punto fijo por estratos siguiendo la aproximación de $HH_{\neg}(\mathcal{C})$ más cercano al enfoque de [122, 74].

Con este sistema concluimos la revisión del estado del arte sobre las capacidades de nuestros sistemas en comparación con otros sistemas de bases de datos. Continuamos presentando el marco $HH_{\neg}(\mathcal{C})$ en el primer capítulo de la memoria.

Capítulo 2

Negación, hipótesis y cuantificadores en bases de datos deductivas con restricciones

En este capítulo presentamos los fundamentos teóricos y la implementación del esquema de bases de datos $HH_{\neg}(\mathcal{C})$. El esquema está basado en la lógica HH (fórmulas de Harrop Hereditarias), un lenguaje más rico que las cláusulas de Horn, dado que incluye cuantificadores en los objetivos así como implicación y disyunción. El lenguaje de base de datos $HH_{\neg}(\mathcal{C})$ combina consultas hipotéticas (derivadas de la implicación), negación, restricciones y también nuevos cuantificadores que no aparecen en otros sistemas de bases de datos deductivas. Para dotar de semántica al lenguaje definimos un cálculo de pruebas, así como una semántica de punto fijo extendiendo técnicas de estratificación propias de bases de datos deductivas. Asimismo probamos que esta semántica operacional es correcta y completa con respecto al cálculo. La semántica de punto fijo guía la implementación de un sistema que hemos desarrollado en SWI-Prolog. A lo largo del capítulo presentamos tres instancias del sistema de restricciones con sus correspondientes resolutores: booleanos, dominios finitos y reales. Hemos integrado además en el sistema funciones de agregación y restricciones de integridad habituales en los sistemas de bases de datos relacionales.

2.1. Introducción

En este capítulo resumimos la investigación que aparece en las publicaciones [A.1, A.2, A.3]. En estas publicaciones presentamos, en primer lugar, el esquema de bases de datos $HH_{\neg}(\mathcal{C})$ y sus ventajas frente a otros sistemas de bases de datos deductivas. También en este resumen se aborda la implementación del sistema basado en el esquema teórico. Como lenguaje para la implementación hemos usado SWI-Prolog [129] y hemos adaptado sus resolutores de restricciones para ajustarlos a las instancias del sistema de restricciones del esquema.

Publicaciones

A continuación hacemos un repaso de los contenidos referentes a $HH_\neg(\mathcal{C})$ que podemos encontrar en cada una de las publicaciones a las que se refiere este capítulo:

- La mayoría de los contenidos del capítulo corresponden al material publicado en [A.3] que describe tanto fundamentos teóricos de $HH_\neg(\mathcal{C})$ como la descripción de su implementación. También en esta publicación encontramos cómo se incorporan las funciones de agregación en el sistema. El cómputo de las funciones de agregación hace uso de la estratificación, que se usa inicialmente para incorporar la negación a las bases de datos deductivas, como podemos ver en la sección 2.3.4.
- Cronológicamente, el primer artículo publicado de esta memoria es [A.1]. Se trata de un artículo que describe el sistema que implementa el esquema $HH_\neg(\mathcal{C})$. En él se pueden encontrar todas las características del sistema, un resumen de los resultados teóricos de la semántica que provee de significado a las bases de datos del lenguaje, así como una descripción de los resolutores que implementan los sistemas de restricciones concretos: booleanos, dominios finitos y reales. Este artículo presenta el núcleo del sistema, que en aquel momento carecía de funciones de agregación y de restricciones de integridad.
- Finalmente, en [A.2] abordamos la incorporación de restricciones de integridad al sistema. Se ha aprovechado nuestro marco de bases de datos deductivas con restricciones (concretamente el uso de las técnicas de estratificación basadas en la construcción de un grafo de dependencias asociado a una base de datos) para añadir esta funcionalidad. Dada la expresividad de nuestro lenguaje, nuestra definición de restricciones de integridad es muy intuitiva y además asegura un funcionamiento correcto en presencia de cómputos locales derivados del cálculo hipotético (veáse la sección 2.3.5).

Contribuciones

A lo largo del capítulo presentamos el nuevo enfoque que supone $\mathrm{HH}_\neg(\mathcal{C})$ como lenguaje de BDD con restricciones y sus aportaciones a este campo. A continuación resumimos estas aportaciones:

- **$HH_\neg(\mathcal{C})$ extiende a la lógica que soporta el lenguaje de base de datos**. En sus comienzos $HH(\mathcal{C})$ [66, 35] carecía de negación. Sin embargo, aportaba recursos expresivos como son el cuantificador universal y la implicación, que no tienen otros lenguajes de programación lógica con restricciones [52]. De la misma forma que surge Datalog a partir de Prolog, $HH_\neg(\mathcal{C})$ surge a partir de $HH(\mathcal{C})$ incorporando la negación para poder aplicarlo al campo de las BDD. Al estar basado en un lenguaje de programación lógica extendido, aporta al campo de las bases de datos las capacidades ya mencionadas heredadas del lenguaje original HH: mayor expresividad, nuevas conectivas y la posibilidad de representar infinitos datos. Además, dado que hemos incorporado negación a $HH_\neg(\mathcal{C})$, el lenguaje es completo con respecto al álgebra relacional que fundamenta las bases de datos relacionales.
- **La ventaja del uso de restricciones**. Nuestro esquema utiliza restricciones, las cuales permiten representar infinitos datos y aportan una gran expresividad y sencillez al desarrollar bases de datos en las que podemos definir intensionalmente la información. Además el lenguaje de restricciones del esquema $HH_\neg(\mathcal{C})$ es más expresivo que el habitual en bases de datos con restricciones.

- De las aportaciones de nuestro lenguaje de bases de datos la más novedosa es **la posibilidad de definir consultas hipotéticas** (denominadas *what-if queries* en el modelo relacional), que son muy útiles para tomar decisiones basadas en datos especulativos. Además aporta sobre el trabajo de Datalog hipotético [15, 16, 13] la posibilidad de que en las consultas aparezcan cuantificadores y la capacidad de incluir restricciones sobre las variables de las cláusulas. El resultado es un lenguaje con mayor expresividad y capaz de representar nuevas bases de datos y consultas que no se pueden definir en otros sistemas.
- **Se ha desarrollado un marco teórico para $HH_\neg(\mathcal{C})$**. Presentamos su semántica de pruebas y su semántica de punto fijo estratificado que sirve de semántica operacional. Además, en [A.3] demostramos la equivalencia entre ambas.
- **El esquema sirve como base para una implementación real**. En la última sección de este capítulo demostramos la utilidad práctica del sistema basado en $HH_\neg(\mathcal{C})$ implementado en SWI-Prolog y presentamos ejemplos reales que procesa este sistema.

El lenguaje de bases de datos $HH_\neg(\mathcal{C})$ y el sistema que lo implementa son los resultados de estas tesis. En las siguientes secciones desarrollamos cada una de las características de $HH_\neg(\mathcal{C})$ que hemos introducido.

El lenguaje $HH_\neg(\mathcal{C})$

Como ya hemos señalado, $HH(\mathcal{C})$ se presentó como un lenguaje de programación lógica. Más adelante, se introdujo la negación, resultando $HH_\neg(\mathcal{C})$, y se demostró que podía ser de gran utilidad dentro de los lenguajes de base de datos debido a algunas características que iremos desgranando a lo largo del capítulo.

Comenzamos abordando algunas nociones como recurso genérico sobre la sintaxis y la semántica pretendida de $HH_\neg(\mathcal{C})$.

Introducimos primero la sintaxis. Para construir elementos de lenguaje $HH_\neg(\mathcal{C})$, necesitamos considerar un conjunto numerable de variables y una signatura que debe contener:

- símbolos de predicados definidos para construir átomos y que representan los nombres de las relaciones de las bases de datos,
- símbolos de predicado predefinidos que incluyen, al menos, el de comparación $\leq$ y un símbolo de predicado de igualdad $\approx$ para construir restricciones atómicas y,
- constantes y símbolos de operación dependientes de un sistema de restricciones concreto para construir términos (junto con las variables).

Formalmente podemos clasificar las fórmulas bien construidas en $HH_\neg(\mathcal{C})$ en cláusulas D (que definirán las relaciones de la base de datos) y objetivos G (que defininirán las consultas a la base de datos). Se definen recursivamente mediante las siguiente reglas:

$$D ::= A \mid G \Rightarrow A \mid D_1 \wedge D_2 \mid \forall x D$$

$$G ::= A \mid \neg A \mid C \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \Rightarrow G \mid C \Rightarrow G \mid \exists x G \mid \forall x G$$

donde A es un átomo, i.e., una fórmula de la forma $p(t_1, \ldots, t_n)$, p es un símbolo de predicado definido con aridad n, y $t_1, \ldots, t_n$ son términos. Para construir términos disponemos de un conjunto de símbolos de constantes y de operaciones, y de un conjunto de variables.

Representamos una restricción con el símbolo C. Veremos la forma de construir restricciones más adelante cuando especifiquemos su sintaxis y las condiciones que tiene que cumplir un sistema de restricciones. Obsérvese que no se permite la negación en la cabeza de una cláusula, solo en su cuerpo.

Las restricciones en el esquema $HH_\neg(\mathcal{C})$

En esta sección presentamos en primer lugar el sistema de restricciones de $HH_\neg(\mathcal{C})$ y, más adelante, las ventajas de su uso en la programación lógica y las bases de datos.

La restricciones que usamos pertenecen a un sistema genérico

$$\mathcal{C} = \langle \mathcal{L}_\mathcal{C}, \vdash_\mathcal{C} \rangle \,,$$

donde $\mathcal{L}_\mathcal{C}$ es el lenguaje de restricciones y $\vdash_\mathcal{C}$ es una *relación de deducibilidad*. $\Gamma \vdash_\mathcal{C} C$ expresa que la restricción C se infiere en el sistema de restricciones mediante la relación $\vdash_\mathcal{C}$ a partir del conjunto de restricciones Γ. Imponemos unas condiciones mínimas a $\mathcal{C}$ para que pueda ser un sistema de restricciones válido:

- $\mathcal{L}_\mathcal{C}$ debe contener, al menos, toda fórmula de primer orden construida usando:
  - $\top$ (*true*), $\bot$ (*false*),
  - símbolos de predicados predefinidos,
  - las conectivas $\wedge, \neg$, y el cuantificador existencial $\exists$.
- Con respecto a $\vdash_\mathcal{C}$:
  - Debe incluir las reglas de inferencia de la lógica intuicionista para las conectivas y cuantificadores que hemos mencionado previamente.
  - Debe ser *compacto*, i.e., $\Gamma \vdash_\mathcal{C} C$ implica que existe un conjunto finito $\Gamma' \subseteq \Gamma$, tal que $\Gamma' \vdash_\mathcal{C} C$.
  - Debe ser *cerrado bajo sustitución*, i.e., $\Gamma \vdash_\mathcal{C} C$ implica que $\Gamma\sigma \vdash_\mathcal{C} C\sigma$ para toda sustitución σ.

La incorporación de la negación (o conectiva $\neg$) al lenguaje HH demanda que la negación se incorpore también en el sistema de restricciones $\mathcal{C}$. Decimos que una restricción C es $\mathcal{C}$-satisfactible si $\emptyset \vdash_\mathcal{C} \overline{\exists} C$, donde $\overline{\exists} C$ representa el cierre existencial de C. C y C' son $\mathcal{C}$-equivalentes si $C \vdash_\mathcal{C} C'$ y $C' \vdash_\mathcal{C} C$.

En los ejemplos, además de las condiciones mínimas incluimos sistemas que incorporan otras conectivas como $\vee$, constantes, operadores aritméticos y más predicados predefinidos ($>$,$\geq$, ...). En concreto, para el sistema de restricciones sobre reales $\mathcal{R}$, $\mathcal{L}_\mathcal{R}$ es el lenguaje de primer orden con todas las conectivas habituales, incluyendo la negación. De esta forma, definimos $\Gamma \vdash_\mathcal{R} C$ cuando $Ax_\mathcal{R} \cup \Gamma \vdash_\approx C$, donde $Ax_\mathcal{R}$ es la axiomatización de Tarski de los números reales [117] y $\vdash_\approx$ es la relación de deducibilidad de la lógica clásica con igualdad. Un ejemplo concreto de restricción dentro de este sistema es $\neg(x \approx 0{,}3)$, que escribimos para simplificar como $x \not\approx 0{,}3$.

La ventaja del uso de restricciones en el contexto de la *programación lógica* es que añaden de forma natural una manera de tratar con conjuntos de infinitos datos usando representaciones finitas. Las bases de datos con restricciones [64] heredan esta capacidad, como veremos en el ejemplo 1.

Para los ejemplos que presentamos a continuación, usamos la instancia $HH_{\neg}(\mathcal{R})$ si estamos usando solo datos reales, sin embargo usamos $HH_{\neg}(\mathcal{FR})$ si usamos valores reales $\mathcal{R}$ junto con otros de dominio finito $\mathcal{F}$. Encontramos más información de este sistema híbrido en [34].

Ejemplo 1 En este ejemplo usamos la instancia $HH_{\neg}(\mathcal{R})$ para describir regiones en el plano. Identificamos una región mediante su función característica (una función booleana que hace corresponder el valor *cierto* a los puntos de la región y *falso* al resto de puntos del plano). Por ejemplo, un rectángulo queda determinado por su esquina inferior izquierda (x_1, y_1) y su esquina superior derecha (x_2, y_2). A continuación, vemos cómo expresar su función característica mediante el uso de cláusulas:

$$\overline{\forall}\ rectangle(x_1, y_1, x_2, y_2, x, y) \Leftarrow x \geq x_1 \wedge x \leq x_2 \wedge y \geq y_1 \wedge y \leq y_2.$$

Podemos representar un rectángulo como un conjunto infinito de puntos de forma finita mediante el uso de restricciones. Desde nuestra perspectiva, si pensamos en la expresividad de las bases de datos, se trata de una característica muy útil dado que nos permite definir fórmulas más complejas que las habituales. A pesar de que las bases de datos fueron concebidas para tratar con datos finitos, mediante restricciones ampliamos su capacidad a los conjuntos (potencialmente) infinitos de datos. Esta es una aportación de $HH_{\neg}(\mathcal{C})$.

El objetivo $rectangle(0, 0, 4, 4, x, y) \wedge rectangle(1, 1, 5, 5, x, y)$ representa la intersección de dos rectángulos, cuya respuesta representamos usando una restricción:

$$(x \geq 1) \wedge (x \leq 4) \wedge (y \geq 1) \wedge (y \leq 4)$$

Por otro lado, un círculo se puede representar usando su centro y su radio, también mediante restricciones no lineales:

$$\overline{\forall}\ circle(xc, yc, r, x, y) \Leftarrow (x - xc) * *2 + (y - yc) * *2 \leq r * *2.$$

Con esta base de datos propuesta podemos consultar si un punto (x, y), que cumple $x^2 + y^2 = 1$ (circunferencia centrada en el origen, de radio 1), se encuentra dentro del círculo con centro $(0, 0)$ y radio 2:

$$\overline{\forall}\ (x^2 + y^2 \approx 1 \Rightarrow circle(0, 0, 2, x, y))$$

En este caso la respuesta es *falso*. Este ejemplo no se puede expresar en otras bases de datos deductivas, dado que, además de restricciones, incluye cuantificador universal e implicación. Tampoco Datalog hipotético [13] ni Datalog con restricciones [95] podrían tratar con este ejemplo dado que utiliza el cuantificador universal. □

Ya que contamos con todos los elementos que lo caracterizan, pasamos a presentar $HH_{\neg}(\mathcal{C})$ como lenguaje de bases de datos.

$HH\neg(\mathcal{C})$ como lenguaje de bases de datos

En la definición de bases de datos $HH_{\neg}(\mathcal{C})$ podemos diferenciar entre los hechos (átomos básicos) que definen la parte *extensional* de la base de datos y las cláusulas, con cabeza y cuerpo, a las que denominamos la parte *intensional* de la base de datos. Esta parte intensional se corresponde con las vistas de las bases de datos relacionales y la parte extensional con las tuplas de una relación.

También establecemos una correspondencia entre los objetivos de los lenguajes lógicos y las consultas a la base de datos. En nuestra propuesta la respuesta a una consulta es una restricción que representa las tuplas o valores que hacen cierta la consulta. Una base de datos, denotada por Δ, es un conjunto de cláusulas.

En primer lugar, mostramos cómo los operadores del AR se pueden expresar mediante predicados en nuestro lenguaje de bases de datos. En la figura 2.1 vemos cómo expresar la proyección, la selección, el producto cartesiano, la unión y la diferencia en $HH_{\neg}(\mathcal{C})$.

- Proyección. $E = \pi_{i_1}, \ldots, \pi_{i_k}(E_1)$

$\overline{\forall}\ e(x_{i_1}, \ldots, x_{i_k}) \Leftarrow e_1(x_1, \ldots, x_n)$.

- Selección. $E = \sigma_{t_1 \theta t_2}(E_1)$

$\overline{\forall}\ e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \wedge C_\theta$.

- Producto Cartesiano. $E = E_1 \times E_2$

$\overline{\forall}\ e(x_1, \ldots, x_n, x_{n+1}, \ldots, x_m) \Leftarrow e_1(x_1, \ldots, x_n) \wedge e_2(x_{n+1}, \ldots, x_m)$.

- Unión. $E = E_1 \cup E_2$

$\overline{\forall}\ e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \vee e_2(x_1, \ldots, x_n)$.

- Diferencia. $E = E_1 - E_2$

$\overline{\forall}\ e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \wedge \neg e_2(x_1, \ldots, x_n)$.

E y E_i son expresiones relacionales expresadas como predicados e y e_i respectivamente. C_θ es la restricción que se corresponde con la condición $t_1 \theta t_2$.

Figura 2.1: Operadores relacionales y sus correspondientes predicados de $HH\neg(\mathcal{C})$.

Para la selección se necesita que el sistema de restricciones $\mathcal{C}$ incorpore el operador θ para poder construir la restricción correspondiente. Por ejemplo, $\sigma_{\$i \leq \$j}$ (selección de todos los t_i con $i \leq j$) se corresponde con $x_i \leq x_j$. Como hemos mencionado en la sección anterior, cualquier sistema de restricciones $\mathcal{C}$ debe incluir al menos $\approx$ y $\leq$ para poder conformar una instancia válida de $HH\neg(\mathcal{C})$.

Ejemplo 2 Como ejemplo de uso de la negación y haciendo referencia al ejemplo 1, definimos la región rayada de la figura 2.2 como el resultado de restar al área del rectángulo externo el rectángulo interno mediante el objetivo:

$$rectangle(0, 0, 4, 4, x, y) \wedge \neg rectangle(1, 1, 3, 3, x, y).$$

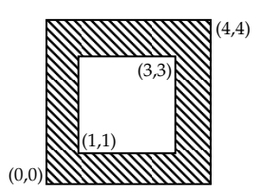


Figura 2.2: Regiones del plano

la respuesta se expresa mediante la restricción:

$$(y > 3 \wedge y \leq 4 \wedge x \geq 0 \wedge x \leq 4) \vee (y \geq 0 \wedge y < 1 \wedge x \geq 0 \wedge x \leq 4) \vee$$
$$(y \geq 0 \wedge y \leq 4 \wedge x > 3 \wedge x \leq 4) \vee (y \geq 0 \wedge y \leq 4 \wedge x \geq 0 \wedge x < 1)$$

Con esta restricción expresamos los datos del área buscada (sobre un conjunto de puntos en el espacio infinitos) con una representación finita. □

En el modelo relacional se trabaja con datos finitos, que se pueden definir directamente como tablas de la base de datos e indirectamente mediante el uso de vistas. Una relación se compone de un nombre y un número determinado de argumentos (su *aridad*). El significado de una relación se corresponde con un conjunto de tuplas.

En $HH\neg(\mathcal{C})$ un predicado tiene también un nombre y una aridad. El significado se corresponde con una conjunto restricción sobre sus argumentos.

Para establecer la correspondencia entre las bases de datos relacionales y las bases de datos $HH\neg(\mathcal{C})$ introducimos un ejemplo que muestra como expresar las mismas relaciones desde ambas aproximaciones relacional y deductiva respectivamente.

Ejemplo 3 En la figura 2.3 definimos extensionalmente algunas relaciones (*client* y *mortgageQuote*), e intensionalmente otra ($accounting$) usando los operadores del AR y también como predicados de $HH_{\neg}(\mathcal{C})$.

(a) Definimos relaciones extensionalmente como tablas:

name	*balance*	*salary*
smith	2000	1200
brown	1000	1500
mcandrew	5300	3000

client

name	*quote*
brown	400
mcandrew	100

mortgageQuote

(b) Ejemplo de relación definida usando los operadores del AR:

$$accounting \leftarrow \pi_{name,salary,quote}(\sigma_{quote \geq 100}(client \bowtie mortgageQuote))$$

(c) Las mismas 3 relaciones anteriores usando el lenguaje de bases de datos $HH_{\neg}(\mathcal{C})$:

$client(smith, 2000, 1200).$ $\quad$ $mortgageQuote(brown, 400).$
$client(brown, 1000, 1500).$ $\quad$ $mortgageQuote(mcandrew, 100).$
$client(mcandrew, 5300, 3000).$

$$\forall name\ \forall salary\ \forall quote\ \forall balance\ (accounting(name, salary, quote) \Leftarrow client(name, balance, salary) \wedge mortgageQuote(name, quote) \wedge quote \geq 100).$$

Figura 2.3: Relaciones de AR y Predicados $HH_{\neg}(\mathcal{C})$

Las relaciones extensionales se definen como tablas en el modelo relacional (a) y como predicados extensionales en nuestro lenguaje de bases de datos (los hechos de (c)). La relación $accounting$ se define usando operadores en el modelo relacional (b) y como predicado

intensional en $HH_{\neg}(\mathcal{C})$. En el ejemplo vemos la correspondencia de los operadores relacionales con las cláusulas de nuestro lenguaje (c).

En el AR el resultado del cómputo de la vista $accounting$ es la siguiente relación:

name	*salary*	*quote*
brown	1500	400
mcandrew	3000	100

$accounting$

En $HH_{\neg}(\mathcal{C})$ esta vista equivale a la consulta $accounting(n, s, q)$ que tiene por respuesta:

$$(n \approx brown \wedge s \approx 1500 \wedge q \approx 400) \vee (n \approx mcandrew \wedge s \approx 3000 \wedge q \approx 100).$$

Un lector familiarizado con modelos de bases de datos deductivas podrá argumentar que nuestro ejemplo puede ser fácilmente trasladado al modelo Datalog con restricciones [58]. Este ejemplo trata de introducir la correspondencia entre AR y $HH_{\neg}(\mathcal{C})$. Más adelante lo extendemos, en el ejemplo 6, para mostrar las nuevas funcionalidades que aporta nuestro lenguaje. □

Consultas hipotéticas

Una de las principales aportaciones de $HH_{\neg}(\mathcal{C})$ es la capacidad de formular consultas hipotéticas. En las bases de datos deductivas el principal referente sobre cómputo de consultas hipotéticas es Bonner [13] y la reciente implementación de Datalog hipótetico [102]. En ambos casos se permite añadir hechos temporalmente a la base de datos.

Presentamos un sencillo ejemplo que demuestra que nuestro lenguaje es tan expresivo como los enfoques anteriores, i.e., un ejemplo donde añadimos temporalmente tuplas a la base de datos en el contexto de una consulta.

Ejemplo 4 Suponemos que tenemos una base de datos con información de las vías de tren construidas entre distintos puntos del mapa, como podemos ver en la figura 2.4.

$railway(madrid, talavera).$
$railway(talavera, navalmoral).$
$railway(navalmoral, caceres).$
$railway(caceres, badajoz).$

Con la siguiente cláusula incluimos todos los puntos que pueden ser unidos mediante las vías del tren como cierre transitivo del predicado $railway$.

$$\overline{\forall}\ railway(x, y) \Leftarrow railway(x, z) \wedge railway(z, y).$$

Ahora supongamos que definimos cuáles de estos puntos tienen estaciones que permiten coger un tren mediante el predicado $station(x)$. Por ejemplo, tenemos $station(madrid)$ y $station(caceres)$. De manera intuitiva podemos definir que se puede viajar a dos puntos siempre que:

- tengamos estación en el origen,
- exista estación en el destino y,
- haya vías que conecten estas dos estaciones.

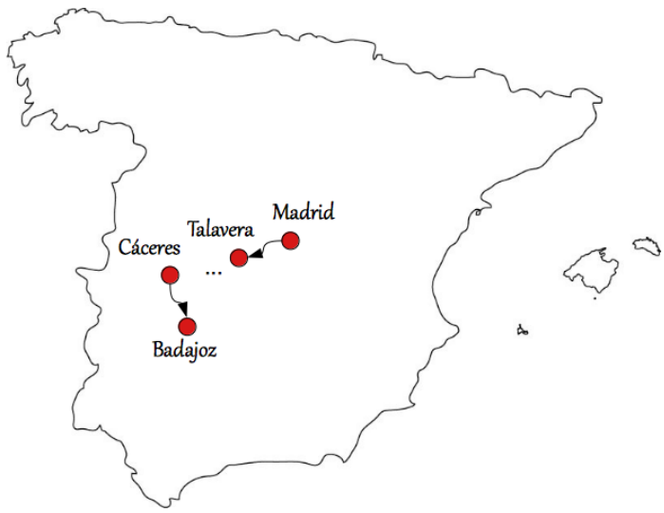


Figura 2.4: Algunos puntos de la base de datos de vías del tren.

Esto se escribe en $\text{HH}_\neg(\mathcal{C})$ como:

$$\overline{\forall}\ travel(x, y) \Leftarrow station(x) \wedge railway(x, y) \wedge station(y).$$

Podemos consultar qué estaciones tendríamos que construir en nuestra red de trenes para viajar desde Madrid a Talavera. Para ello formulamos la consulta:

$$station(x) \Rightarrow travel(madrid, talavera)$$

La restricción respuesta es $x \approx talavera$. □

Expresividad de $\text{HH}_\neg(\mathcal{C})$

Una vez introducidas la sintaxis y las características de $HH_\neg(\mathcal{C})$, introducimos más ejemplos que demuestran las ventajas de nuestro lenguaje de bases de datos frente a los habituales. Más concretamente, veremos ejemplos en los que manejamos de forma integrada:

- la capacidad de formular consultas hipotéticas,
- el uso del cuantificador existencial y el universal,
- la capacidad de proporcionar resultados usando restricciones.

Para los siguientes ejemplos volvemos a usar una instancia de $HH_\neg(\mathcal{C})$ que combina restricciones de dominio finito y real, i.e., $HH_\neg(\mathcal{FR})$.

Ejemplo 5 Utilizamos la siguiente base de datos para una compañía aérea, compuesta por el predicado $flight(Origin, Destination, Time)$ que representa la base de datos extensional de vuelos directos desde $Origin$ hasta $Destination$ y con duración $Time$:

$$flight(mad, par, 1{,}5).$$
$$flight(par, ny, 10).$$
$$flight(london, ny, 9).$$

Además, $travel(Origin, Destination, Time)$ representa la base de datos intensional. La idea tras esta relación es la capacidad de viajar desde $Origin$ hasta $Destination$ si disponemos de un tiempo $Time$, con la posibilidad de concatenar más de un vuelo.

$$\overline{\forall}\ travel(x, y, t) \Leftarrow flight(x, y, w) \wedge t \geq w.$$
$$\overline{\forall}\ travel(x, y, t) \Leftarrow flight(x, z, t_1) \wedge travel(z, y, t_2) \wedge t \geq t_1 + t_2.$$

Comenzamos con las consultas de la base de vuelos. Por ejemplo en $HH_{\neg}(\mathcal{C})$ se puede consultar cuál es la duración que debe tener un vuelo desde Madrid hasta Londres para poder viajar desde Madrid a Nueva York en un tiempo de a lo sumo 11 horas.

$$flight(mad, london, t) \Rightarrow travel(mad, ny, 11)$$

La respuesta es la restricción $11 \geq t + 9$ que es $\mathcal{FR}$-equivalente a la respuesta $t \leq 2$.

Otro ejemplo de consulta hipotética es preguntar si se puede volar desde Madrid hacia algún sitio en un tiempo mayor que 1,5 horas. El objetivo

$$\forall t(t > 1{,}5 \Rightarrow \exists y\ travel(mad, y, t))$$

es además un ejemplo de uso de cuantificación universal y de uso explícito del cuantificador $\exists$ para no devolver una respuesta concreta para y. La respuesta a esta consulta es $\top$.

Comparando $HH_{\neg}(\mathcal{C})$ con el cálculo relacional, podemos formular la consulta

$$\neg(\exists t\ flight(x, y, t)) \wedge x \not\approx y$$

o su equivalente $(\forall t\ \neg flight(x, y, t)) \wedge x \not\approx y$, para determinar las ciudades sin vuelos directos entre ellas. Esta fórmula no es *segura* en el cálculo relacional de dominios [27] dado que contiene una fórmula negada cuyas variables libres no están limitadas. En $HH\neg(\mathcal{C})$ esto se delega en el sistema de restricciones de la instancia concreta como veremos más adelante en las secciones teóricas.

De hecho $(\forall t\ \neg flight(x, y, t)) \wedge x \not\approx y$ es una consulta válida en $HH_{\neg}(\mathcal{FR})$ que tiene por respuesta la restricción:

$$(x \not\approx mad \vee y \not\approx par) \wedge (x \not\approx par \vee y \not\approx ny) \wedge (x \not\approx lon \vee y \not\approx ny)$$

en el domino de ciudades de la base de datos. De nuevo, se trata de una consulta que no puede ser formulada en Datalog, en este caso debido al cuantificador universal.

Supongamos la situación más realista de que los vuelos puedan retrasarse. Un retraso en el vuelo entre x e y de un tiempo d [1] se representa mediante el predicado $delay(x, y, d)$. A continuación definimos el predicado $itinerary$ para representar los posibles viajes en la base de datos teniendo en cuenta los retrasos mediante $delay$:

$$\overline{\forall}\ itinerary(x, y, t, 0) \Leftarrow flight(x, y, t) \wedge \neg delay(x, y, d).$$
$$\overline{\forall}\ itinerary(x, y, t, d) \Leftarrow flight(x, y, t_1) \wedge delay(x, y, d) \wedge t \geq t_1 + d.$$
$$\overline{\forall}\ itinerary(x, y, t, d) \Leftarrow itinerary(x, z, t_1, d_1) \wedge itinerary(z, y, t_2, d_2) \wedge$$
$$\wedge\ t \geq t_1 + t_2 \wedge d = d_1 + d_2$$

Al igual que en la relación travel, t representa un valor mayor o igual a la duración total del itinerario y d el retraso acumulado. Las tuplas de *delay* pueden estar en la base de datos extensional o bien se pueden asumir en una consulta, como por ejemplo en:

$$\forall x(delay(par, x, 1) \wedge delay(mad, par, 0{,}5)) \Rightarrow itinerary(mad, ny, t, d).$$

[1] Por simplicidad en este ejemplo suponemos que no hay más de un vuelo con el mismo origen y destino.

que representa el tiempo necesario para volar desde Madrid a Nueva York, asumiendo que cualquier vuelo desde París tiene un retraso de una hora, y además, el vuelo de Madrid a París se retrasa media hora.

Para resolver esta consulta, la cláusula

$$\forall x(delay(par, x, 1) \wedge delay(mad, par, 0{,}5))$$

se añade localmente a la base de datos y se descarta tras el cómputo. □

Ejemplo 6 A continuación extendemos la base de datos para un banco del ejemplo 3. La base de datos extensional viene dada mediante las relaciones que dan información de los clientes y de su cuota hipotecaria en euros:

$\%client(Name, Balance, Salary).$ $\quad$ $\%mortgageQuote(Name, Quote).$
$client(smith, 2000, 1200).$ $\quad$ $mortgageQuote(brown, 400).$
$client(brown, 1000, 1500).$ $\quad$ $mortgageQuote(mcandrew, 100).$
$client(mcandrew, 5300, 3000).$

También tenemos información extensional de las deudas pendientes y las oficinas asignadas a cada cliente:

$\%branch(Office, Name).$ $\quad$ $\%pastDue(Name, Amount).$
$branch(lon, smith).$ $\quad$ $pastDue(smith, 3000).$
$branch(mad, brown).$ $\quad$ $pastDue(mcandrew, 100).$
$branch(par, mcandrew).$

Para simplificar añadimos implícitamente la restricción adicional de que un cliente puede tener a lo sumo una cuota hipotecaria.

La primera relación de la parte intensional de la base de datos representa los clientes que tienen asignada una cuota hipotecaria.

$$\overline{\forall}\ hasMortgage(x) \Leftarrow mortgageQuote(x, y).$$

La siguiente relación nos informa de los clientes en números rojos, como aquéllos cuya deuda es mayor que su saldo.

$$\overline{\forall}\ debtor(x) \Leftarrow client(x, y, z) \wedge pastDue(x, w) \wedge w > y.$$

Con la siguiente relación se determina la tasa de interés que se aplicará a cada cliente:

$$\overline{\forall}\ interestRate(x, 2) \Leftarrow client(x, y, z) \wedge y < 1200.$$
$$\overline{\forall}\ interestRate(x, 5) \Leftarrow client(x, y, z) \wedge y \geq 1200.$$

Usamos la relación $newMortgage(Name, Quote)$ para ampliar la hipoteca con una nueva cuota $Quote$ a clientes $Name$ que no tienen un saldo negativo y verifican alguna de las dos condiciones: si no tiene ya una hipoteca o bien si su nueva cuota no es superior al 40 % de su sueldo. En general, no se concederá una nueva hipoteca si su cuota supera el 40 % del salario del cliente.

$$\begin{aligned}\overline{\forall}\ newMortgage(x, w) \Leftarrow\ & client(x, y, z) \wedge \neg debtor(x) \wedge \\ & \neg hasMortgage(x) \wedge w \leq 0{,}4 * z.\\ \overline{\forall}\ newMortgage(x, w) \Leftarrow\ & client(x, y, z) \wedge \neg debtor(x) \wedge \\ & mortgageQuote(x, w') \wedge w + w' \leq 0{,}4 * z.\end{aligned}$$

Además definimos una relación que incluye los clientes que tienen una hipoteca.

$$\overline{\forall}\ gotMortgage(x) \Leftarrow newMortgage(x, w).$$

Si cumple los requisitos para una nueva hipoteca se le puede dar un crédito personal de hasta 6.000. O bien, en caso contrario esta cantidad asciende a un intervalo entre 6.000 y 20.000, dado que involucra menos riesgo. La relación $personalCredit(Name, Amount)$ formaliza de forma sencilla todas las condiciones que acabamos de imponer.

$$\overline{\forall}\ personalCredit(x, y) \Leftarrow (gotMortgage(x) \wedge y < 6000) \vee (\neg gotMortgage(x) \wedge y \geq 6000 \wedge y < 20000).$$

Definimos un nuevo predicado del lenguaje que nos dará información del sueldo de clientes con cuota hipotecaria superior a 100 mediante la relación $accounting(Name, Salary, Quote)$ que es la misma $accounting$ del ejemplo 3.

$$\overline{\forall}\ accounting(x, z, w) \Leftarrow client(x, y, z) \wedge mortgageQuote(x, w) \wedge w \geq 100.$$

A continuación mostramos ejemplos de consultas. Un primer ejemplo sencillo sería consultar si todos los clientes están en números rojos.

$$\forall x\ debtor(x).$$

cuya respuesta obvia es $\bot$.

La existencia de deudores con un descubierto superior a 1.000 se pueden averiguar con:

$$\exists x\ \exists y\ debtor(x) \wedge pastDue(x, y) \wedge y > 1000.$$

y la respuesta es $\top$. Estamos usando cuantificadores sobre las variables x e y, dado que no queremos respuesta explícita sobre ellas. En otro caso obtendríamos como respuesta una restricción sobre estas variables.

La siguiente consulta devuelve la tasa de interés de un cliente cualquiera si este tiene un balance mayor que 2.000.

$$\forall x\ \exists y\ \exists z\ (client(x, y, z) \Rightarrow (y > 2000 \Rightarrow interestRate(x, w))).$$

En este ejemplo usamos una implicación anidada para formular una consulta hipotética cuya respuesta es la restricción $w \approx 5$.

Usamos la conectiva $\neg$ para preguntar a qué clientes se les puede conceder una hipoteca de 400 pero no un crédito.

$$newMortgage(x, 400) \wedge \neg personalCredit(x, y).$$

y la respuesta es $x \approx mcandrew \wedge y \geq 6000 \wedge y < 20000$, que quiere decir que podríamos concederle la hipoteca solamente al cliente McAndrew pero no un crédito entre 6.000 y 20.000. □

Con este ejemplo concluimos la presentación sobre las posibilidades expresivas que ofrece $HH\neg(\mathcal{C})$ que no aparecen en otros lenguajes de bases de datos. A continuación abordamos los fundamentos teóricos del esquema y su implementación.

2.2. Fundamentos teóricos de $HH\neg(\mathcal{C})$

En esta sección presentamos cómo se han adaptado los formalismos previos para dar fundamento teórico al esquema $HH_\neg(\mathcal{C})$. Los resultados que aparecen en esta sección, así como sus demostraciones, se pueden encontrar en [A.3]. Hemos definido dos semánticas para nuestro lenguaje de bases de datos: una semántica de pruebas y una semántica de punto fijo. Además, terminamos esta sección demostrando que ambas son equivalentes.

Para el lenguaje $HH(\mathcal{C})$ se habían definido anteriormente una semántica de pruebas [66] y una semántica de punto fijo [35] que sentaron las bases del esquema. En esta sección presentamos cómo se incorpora la negación al esquema. Al introducir la negación en el lenguaje surge el problema de asegurar la existencia de un único *modelo mínimo* para una base de datos. Como hemos señalado en el capítulo anterior, este problema se ha abordado en en el campo de la programación lógica mediante distintas propuestas [21, 2, 107, 37, 124, 100]. Sin embargo, en este trabajo el manejo de la negación cobra especial importancia dado que:

- Hemos incluido la negación en el lenguaje para que sea completo con respecto al AR.
- $HH_\neg(\mathcal{C})$ maneja consultas hipotéticas que conllevan cómputos locales debido a los cambios temporales que se producen en la base de datos al introducir hipótesis, y por tanto es necesario adaptar técnicas conocidas en el campo deductivo como el grafo de dependencias y la estratificación para poder asegurar un cómputo correcto (como vemos en la sección 2.2.2).

2.2.1. Semántica de pruebas

El cálculo que fundamenta la semántica denotacional de $HH\neg(\mathcal{C})$ se denomina $\mathcal{UC}_\neg$ (por sus siglas en inglés *Uniform Calculus handling Constraints and Negation*). Se trata de un cálculo de secuentes que surge al añadir la negación al cálculo $\mathcal{UC}$ que se introdujo en [66] para formalizar $HH(\mathcal{C})$.

$\mathcal{UC}_\neg$ combina reglas de inferencia de la lógica intuicionista con la relación de deducibilidad $\vdash_\mathcal{C}$ de un sistema de restricciones genérico $\mathcal{C}$. La idea es que una consulta G será cierta para una base de datos Δ si la restricción C se satisface. El cálculo $\mathcal{UC}_\neg$ lleva a cabo solamente demostraciones uniformes en el sentido de Miller *et. al.* [75], i.e., demostraciones orientadas a los objetivos. Las reglas del cálculo aparecen en la figura 2.5.

La notación $\Delta; \Gamma \vdash_{\mathcal{UC}_\neg} G$ denota que el secuente $\Delta; \Gamma \vdash G$ se prueba usando las reglas de $\mathcal{UC}_\neg$. En general, si $\Delta; C \vdash_{\mathcal{UC}_\neg} G$, entonces C se denomina restricción respuesta a la consulta G en la base de datos Δ, y se identifica con la respuesta de una consulta G que formulamos a dicha base de datos Δ. Los secuentes tienen la forma $\Delta; \Gamma \vdash G$, donde las bases de datos Δ y los conjuntos de restricciones Γ están a la izquierda y las consultas a la derecha. Una demostración de un secuente es un árbol finito. La raíz del árbol es el secuente que queremos probar, los nodos internos son también secuentes que son instancias de la conclusión de una regla del cálculo siendo sus hijos las premisas de dicha regla, mientras que los nodos hoja son de la forma $\Gamma \vdash_\mathcal{C} C$.

En la figura 2.5 utilizamos la noción de *elaboración* de un programa Δ (véase su definición en la sección 3.1.2 de [A.3]). Las cláusulas elaboradas son fórmulas de la forma $\forall x_1 \ldots \forall x_n (G \Rightarrow A)$. Sin embargo, las cláusulas dentro de G no tienen que estar elaboradas. Además si A' y A son átomos de la forma $p(t'_1, \ldots, t'_n)$ y $p(t_1, \ldots, t_n)$, respectivamente, $A' \approx A$ representa la restricción $t'_1 \approx t_1 \wedge \ldots \wedge t'_n \approx t_n$.

$$\frac{\Gamma \vdash_{\mathcal{C}} C}{\Delta; \Gamma \vdash C}\ (C) \qquad \frac{\Delta; \Gamma \vdash \exists x_1 \ldots \exists x_n((A' \approx A) \wedge G)}{\Delta; \Gamma \vdash A}\ (Clause)\ (*), \text{ donde}$$

$\forall x_1 \ldots \forall x_n(G \Rightarrow A')$ es una variante de una fórmula que aparece en $elab(\Delta)$

$$\frac{\Delta; \Gamma \vdash G_i}{\Delta; \Gamma \vdash G_1 \vee G_2}\ (\vee)\ (i = 1, 2) \qquad \frac{\Delta; \Gamma \vdash G_1 \quad \Delta; \Gamma \vdash G_2}{\Delta; \Gamma \vdash G_1 \wedge G_2}\ (\wedge)$$

$$\frac{\Delta, D; \Gamma \vdash G}{\Delta; \Gamma \vdash D \Rightarrow G}\ (\Rightarrow) \qquad \frac{\Delta; \Gamma, C \vdash G}{\Delta; \Gamma \vdash C \Rightarrow G}\ (\Rightarrow_C)$$

$$\frac{\Delta; \Gamma, C \vdash G[y/x] \quad \Gamma \vdash_{\mathcal{C}} \exists y C}{\Delta; \Gamma \vdash \exists x G}\ (\exists)(**) \qquad \frac{\Delta; \Gamma \vdash G[y/x]}{\Delta; \Gamma \vdash \forall x G}\ (\forall)(**)$$

$$\frac{\Gamma \vdash_{\mathcal{C}} \neg C \quad \text{para todo } \Delta; C \vdash A}{\Delta; \Gamma \vdash \neg A}\ (\neg)$$

$(*)$ $x_1, \ldots, x_n$ frescas para A

$(**)$ y frescas para las fórmulas en la conclusión de la regla

Figura 2.5: Reglas para el cálculo de secuentes $\mathcal{UC}_\neg$

La incorporación de la negación hace necesaria la extensión de la noción de derivabilidad. Con la regla $(\neg)$ se formaliza la derivación de átomos negados. Para interpretar una consulta $\neg A$ para una base de datos Δ se obtiene una restricción respuesta C. Si C' es una respuesta posible a la consulta A sobre Δ, entonces $C \vdash_{\mathcal{C}} \neg C'$. Consideramos $(\neg)$ una *metarregla* dado que su premisa toma todas las derivaciones de la forma $\Delta; C \vdash A$ del átomo A. En la práctica hay una derivación para $\neg A$ cuando el conjunto de restricciones respuesta de A, sobre Δ, es finito. A continuación mostramos un ejemplo de árbol de derivación para la regla de la negación y terminaremos la sección abordando la terminación de este cálculo.

Ejemplo 7 Volviendo a los ejemplos 1 y 2, sea Δ el conjunto:

$\{\overline{\forall}\ (x \geq x_1 \wedge x \leq x_2 \wedge y \geq y_1 \wedge y \leq y_2 \Rightarrow rectangle(x_1, y_1, x_2, y_2, x, y))\}$,

y $G \equiv rectangle(0, 0, 4, 4, x, y), \neg rectangle(1, 1, 3, 3, x, y)$. La restricción respuesta

$$\begin{aligned} C \equiv\ & ((y > 3) \wedge (y \leq 4) \wedge (x \geq 0) \wedge (x \leq 4)) \vee \\ & ((y \geq 0) \wedge (y < 1) \wedge (x \geq 0) \wedge (x \leq 4)) \vee \\ & ((y \geq 0) \wedge (y \leq 4) \wedge (x > 3) \wedge (x \leq 4)) \vee \\ & ((y \geq 0) \wedge (y \leq 4) \wedge (x \geq 0) \wedge (x < 1)) \end{aligned}$$

se puede obtener mediante la siguiente deducción:

$$\dfrac{\dfrac{\dfrac{C \vdash_{\mathcal{R}} \exists a_1\ \exists a_2\ \exists b_1\ \exists b_2\ \exists x_1\ \exists y_1(a_1 \approx 0 \wedge x_1 \approx x \wedge \ldots)}{\Delta; C \vdash \exists a_1\ \exists a_2\ \exists b_1\ \exists b_2\ \exists x_1\ \exists y_1(a_1 \approx 0 \wedge x_1 \approx x \wedge x_1 \geq a_1 \wedge \\ a_2 \approx 0 \wedge y_1 \approx y \wedge x_1 \leq b_1 \wedge b_1 \approx 4 \wedge y_1 \geq a_2 \wedge b_2 \approx 4 \wedge y_1 \leq b_2)}\ (C)}{\Delta; C \vdash rectangle(0, 0, 4, 4, x, y)}\ (Clause) \quad \mathbf{D}}{\Delta; C \vdash rectangle(0, 0, 4, 4, x, y) \wedge \neg rectangle(1, 1, 3, 3, x, y)}\ (\wedge)$$

donde **D** es una deducción para $\Delta; C \vdash \neg rectangle(1, 1, 3, 3, x, y)$ cuyos últimos pasos tienen la forma:

$$\dfrac{C \vdash_{\mathcal{R}} \neg(x \geq 1 \wedge y \geq 1 \wedge x \leq 3 \wedge y \leq 3) \qquad \dfrac{< \textit{resto de la derivación} >}{\Delta;\ x \geq 1 \wedge y \geq 1 \wedge x \leq 3 \wedge y \leq 3 \vdash rectangle(1, 1, 3, 3, x, y)}}{\Delta; C \vdash \neg rectangle(1, 1, 3, 3, x, y)}\ (\neg)$$

Para asegurar un proceso de resolución de objetivos correcto y completo debemos imponer algunas condiciones de finitud que hacen viable la metarregla ($\neg$).

Se debe garantizar que el conjunto de respuestas para un átomo (que aparece negado dentro de un objetivo) se pueda calcular en un número finito de pasos. En concreto hay que garantizar que no hay infinitas restricciones respuesta para un átomo. Para ello es necesario imponer unas condiciones de terminación a los sistemas de restricciones C (similares a las condiciones de seguridad que aparecen definidas en [96]). Al imponer unas condiciones de compacidad a C garantizamos que se pueda representar la respuesta en en lenguaje de C, mediante un conjunto finito de restricciones y, que por tanto, el cómputo termina si se garantiza monotonía.

El uso de la estratificación es una técnica adecuada para garantizar monotonía en presencia de la negación. Además otra ventaja del uso de la estratificación es que se puede combinar de forma sencilla con nuestra noción de sistema de restricciones lo que lleva a una semántica operacional para $HH_{\neg}(\mathcal{C})$ que dota de significado a toda la base de datos y tiene en cuenta condiciones de seguridad para la terminación del cómputo.

A continuación explicamos la semántica de punto de fijo de $HH_{\neg}(\mathcal{C})$ que fundamenta la implementación del sistema y terminamos demostrando la equivalencia entre la semántica de pruebas presentada aquí y la de punto fijo.

2.2.2. Semántica de punto fijo

Al igual que hemos hecho con la semántica de pruebas, en esta sección presentamos las principales aportaciones del trabajo frente a lo anteriormente publicado en [35] para interpretar las bases de datos $HH_{\neg}(\mathcal{C})$. La semántica original de punto fijo de $HH(\mathcal{C})$ estaba basada en una *relación de forzado* entre programas, conjuntos de restricciones y objetivos que establecían si una interpretación hacía cierto un objetivo G, el contexto $\langle \Delta, \Gamma \rangle$ de un programa Δ y un conjunto de restricciones Γ. Seguimos una aproximación similar a la que aparece en el capítulo 3 de [122], dado que usamos técnicas de estratificación de una base de datos que se basa en la definición de un grafo de dependencias.

Introducimos informalmente las *interpretaciones* como funciones que se aplican a cada base de datos Δ y devuelven como resultado conjuntos de pares (compuestos por un átomo y su restricción asociada). Las interpretaciones dependen siempre del contexto al que se aplican dado que al calcular consultas con implicación o bien Δ o bien Γ pueden aumentar localmente con el antecedente de la implicación. En nuestro esquema las interpretaciones son funciones que dan significado a toda la base de datos devolviendo conjuntos de pares (A, C).

Grafo de dependencias y estratificación

Dado un conjunto de cláusulas y objetivos Φ, el grafo de dependencias correspondiente, DG_{Φ}, es un grafo dirigido tal que:

- los *nodos* son los símbolos de predicados definidos en Φ,
- y los *arcos* vienen determinados por los símbolos de implicación de las fórmulas.

Como hemos dicho, cuando construimos el grafo de dependencias, debemos tener en cuenta que las implicaciones pueden aparecer dentro de un objetivo, y por tanto, en el cuerpo de una cláusula. Una implicación de la forma $F1 \Rightarrow F2$ produce arcos en nuestro grafo desde los símbolos de predicado definidos que aparecen dentro de $F1$ hacia cada símbolo de predicado definido que aparece dentro de $F2$.

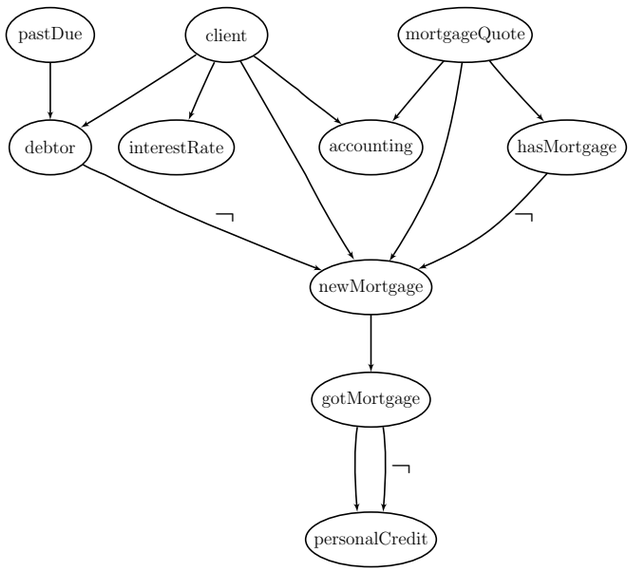


Figura 2.6: Grafo de dependencias del ejemplo 6

Los arcos pueden estar *etiquetados negativamente* (y los representamos con el símbolo ¬) si el átomo correspondiente aparece negado a la izquierda de la implicación. En el caso de las restricciones, dado que no contienen símbolos de predicados definido, no producen este tipo de dependencias.

Ejemplo 8 Sea Δ la base de datos para el banco del ejemplo 6. En la figura 2.6 se muestra el grafo de dependencias para Δ (menos el predicado $branch$ que se representaría como un nodo aislado). □

A continuación formalizamos la definición de dependencias entre predicados.

Definición 1 Dado un conjunto de fórmulas Φ, su correspondiente grafo de dependencias DG_Φ, y dos predicados cualesquiera p y q, se dice que:

- q *depende* de p si hay un camino desde p hasta q en DG_Φ.
- q *depende negativamente* de p si hay un camino desde p hasta q en DG_Φ con, al menos, un arco etiquetado negativamente. □

Partiendo de la definición de dependencias, definimos la noción de estratificación para un conjunto de predicados. Utilizamos esta noción siguiendo la aproximación de [122] para asegurar que el significado de un predicado esté completamente calculado antes de aplicar la negación sobre él, i.e., por ejemplo en el programa $p(x) \Leftarrow \neg q(x)$ el significado de q debe ser calculado antes que el significado de p.

Definición 2 Sea Φ un conjunto de fórmulas y $P = \{p_1, \ldots, p_n\}$ el conjunto de los símbolos de predicados definidos en Φ. Una *estratificación* de Φ es una función $s : P \to \{1, \ldots, n\}$ tal que $s(p) \leq s(q)$ si q depende de p, y $s(p) < s(q)$ si q depende negativamente de p. Decimos que Φ es estratificable si podemos encontrar una estratificación para él.

Ejemplo 9 Una estratificación para la base de datos Δ del ejemplo 5 incluirá todos los predicados dentro del estrato 1 menos $nondeltravel$ y $trip$, que pertenecerán al estrato 2.

Intuitivamente, este hecho nos muestra que para evaluar $nondeltravel$, el resto de predicados (menos $trip$) deberían haber sido evaluados previamente (en particular $delayed$). Cuando se formula la consulta:

$$G \equiv \exists t\ deltravel(x, y, t) \Rightarrow delayed(x, y),$$

el conjunto aumentado $\Delta \cup \{G\}$ continúa siendo estratificable. Sin embargo, si se formula

$$G' \equiv trip(mad, lon, T) \Rightarrow delay(mad, ny, t),$$

el conjunto extendido $\Delta \cup \{G'\}$ será no estratificable. Este hecho se debe a que G' añade la dependencia $trip \rightarrow delay$, y consecuentemente, cualquier estratificación s debe satisfacer $s(trip) \leq s(delay) \leq s(delayed) < s(nondeltravel) \leq s(trip)$, lo cual no es posible. □

En adelante, suponemos que existe una estratificación s para el conjunto $\Delta \cup \{G\}$. También usamos la noción de estrato de un átomo (estrato de su símbolo de predicado). Finalmente mostramos la forma de extender esta noción a cualquier fórmula, o conjunto de fórmulas, siguiendo la siguiente definición:

Definición 3 Sea F un objetivo o una cláusula. El *estrato de una fórmula* F, denominado $str(F)$, se define recursivamente como:

$str(C) = 1$
$str(p(t_1, \ldots, t_n)) = s(p)$
$str(\neg A) = 1 + str(A)$
$str(F_1 \square F_2) = max(str(F_1), str(F_2))$, donde $\square \in \{\wedge, \vee, \Rightarrow\}$
$str(QxF) = str(F)$, donde $Q \in \{\exists, \forall\}$

Además, el *estrato de un conjunto de fórmulas* Φ es $str(\Phi) = max\{str(F) \mid F \in \Phi\}$. □

Todas estas definiciones, junto con sus explicaciones y otros ejemplos se pueden encontrar también en la Sección 5 de [A.3].

Interpretaciones estratificadas y relación de forzado

Sea $\mathcal{W}$ el conjunto de bases de datos estratificables con respecto a una estratificación fija s. Las interpretaciones y el operador de punto fijo se aplican sobre las distintas interpretaciones de bases de datos de $\mathcal{W}$. Estas interpretaciones se calculan estrato por estrato mediante el uso de un operador de punto fijo que definimos más adelante (siguiendo una aproximación similar a [122]).

Sea At el conjunto de átomos abiertos, i.e., símbolos de predicado de una signatura aplicados a variables; y sea $\mathcal{SL}_\mathcal{C}$ el conjunto de fórmulas $\mathcal{C}$-satisfactibles módulo $\mathcal{C}$-equivalencia. El conjunto $At \times \mathcal{SL}_\mathcal{C}$ es finito dado que consideramos signaturas finitas y sistemas de restricciones compactos. Una interpretación sobre un estrato i para una base de datos pertenecerá al conjunto de pares $(A, [C]) \in At \times \mathcal{SL}_\mathcal{C}$, donde $str(A) \leq i$ y $[C]$ es el conjunto de restricciones $\mathcal{C}$-equivalentes a C.

Definimos formalmente una interpretación.

Definición 4 Sea $i \geq 1$. Una *interpretación I sobre un estrato i* es una función

$$I : \mathcal{W} \rightarrow \mathcal{P}(At \times \mathcal{SL}_\mathcal{C}),$$

tal que, para todo $\Delta \in \mathcal{W}$, si $(A, [C]) \in I(\Delta)$ entonces $str(A) \leq j$. Denotamos con $\mathcal{I}_i$ al *conjunto de interpretaciones sobre i*. □

Para simplificar utilizamos la siguiente notación:

- $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, en vez de $(A, [C])$: asumiendo que este C es cualquier restricción que represente a su clase de equivalencia $[C]$.
- $[I(\Delta)]_i$ representa $\{(A, C) \in I(\Delta) \mid str(A) = i\}$.

Nótese que, si $str(\Delta) = k$, entonces $\{[I(\Delta)]_i \mid 1 \leq i \leq k\}$ es una partición de $I(\Delta)$. Para cada $i \geq 1$, definimos un orden sobre $\mathcal{I}_i$ como:

Definición 5 Sea $i \geq 1$ y $I_1, I_2 \in \mathcal{I}_i$. I_1 *es menor o igual que* I_2 *en el estrato* i, denotado por $I_1 \sqsubseteq_i I_2$, siempre que para todo $\Delta \in \mathcal{W}$ se satisfagan las siguientes condiciones:

- $[I_1(\Delta)]_j = [I_2(\Delta)]_j$, para todo $1 \leq j < i$.
- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$. □

Es sencillo concluir que para todo $i \geq 1$, $(\mathcal{I}_i, \sqsubseteq_i)$ es un retículo. La idea tras esta definición es, que cuando una interpretación sobre un estrato i aumenta, la información del estrato inferior permanece invariable. De forma que, si $str(\neg A) = i$, dado que $str(A) = i - 1$, el significado de A en el estrato i no cambia y con ello podemos garantizar la monotonía incluso para átomos negados.

Lema 1 Para cualquier $i \geq 1$, toda cadena de interpretaciones $(\mathcal{I}_i, \sqsubseteq_i)$, $\{I_n\}_{n \geq 0}$, tal que $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$, tiene un supremo $\bigsqcup_{n \geq 0} I_n$, que definimos como $(\bigsqcup_{n \geq 0} I_n)(\Delta) = \bigcup \{I_n(\Delta) \mid n \geq 0\}$, para todo $\Delta \in \mathcal{W}$.

La siguiente definición formaliza la noción de que una interpretación I hace cierta la consulta G en el contexto de una base de datos Δ, si se satisface la restricción C. Como hemos anticipado, asumimos que s es una estratificación válida para Δ y también para la base de datos extendida $\Delta \cup \{G\}$.

Definición 6 Sea $i \geq 1$. La *relación de forzado* $\Vdash$ entre pares I, Δ y pares (G, C) (donde $I \in \mathcal{I}_i$, $str(G) \leq i$, y C es $\mathcal{C}$-satisfactible) se define recursivamente mediante las reglas siguientes. Cuando $I, \Delta \Vdash (G, C)$, decimos que (G, C) es *forzado* por I en el contexto de Δ.

- $I, \Delta \Vdash (C', C) \iff C \vdash_{\mathcal{C}} C'$.
- $I, \Delta \Vdash (A, C) \iff (A, C) \in I(\Delta)$.
- $I, \Delta \Vdash (\neg A, C) \iff$ para cada $(A, C') \in I(\Delta)$, es cierto que $C \vdash_{\mathcal{C}} \neg C'$. Si no existen pares de la forma (A, C') en $I(\Delta)$, entonces $C \equiv \top$.
- $I, \Delta \Vdash (G_1 \wedge G_2, C) \iff$ para cada $i \in \{1, 2\}$, $I, \Delta \Vdash (G_i, C)$.
- $I, \Delta \Vdash (G_1 \vee G_2, C) \iff$ para algún $i \in \{1, 2\}$ $I, \Delta \Vdash (G_i, C)$.
- $I, \Delta \Vdash (D \Rightarrow G, C) \iff I, \Delta \cup \{D\} \Vdash (G, C)$.
- $I, \Delta \Vdash (C' \Rightarrow G, C) \iff I, \Delta \Vdash (G, C \wedge C')$.
- $I, \Delta \Vdash (\exists x G, C) \iff$ existe C' tal que $I, \Delta \Vdash (G[y/x], C')$, donde y no aparece libre en Δ, $\exists x G$, C, y $C \vdash_{\mathcal{C}} \exists y C'$.
- $I, \Delta \Vdash (\forall x G, C) \iff I, \Delta \Vdash (G[y/x], C)$, donde y no aparece libre en Δ, $\forall x G$, C. □

La significado de la interpretación de un estrato viene dado por el menor punto fijo de un operador continuo que transforma interpretaciones y que definimos a continuación:

Definición 7 Sea $i \geq 1$ un estrato. El *operador* $T_i : \mathcal{I}_i \longrightarrow \mathcal{I}_i$ transforma interpretaciones sobre i como sigue. Para $I \in \mathcal{I}_i$, $\Delta \in \mathcal{W}$ y $(A, C) \in At \times \mathcal{SL}_C$, se tiene $(A, C) \in T_i(I)(\Delta)$ cuando:

- $(A, C) \in [I(\Delta)]_j$ para algún $j < i$, o
- $str(A) = i$ y hay una variante $\forall x_1 \ldots \forall x_n(G \Rightarrow A')$ de una cláusula en $elab(\Delta)$, tal que las variables $x_1 \ldots x_n$ no aparecen libres en A y se cumple que:

$$I, \Delta \models (\exists x_1 \ldots \exists x_n(A \approx A' \wedge G), C). \qquad \square$$

Un aspecto importante del operador T_i es que, para una base de datos Δ, T_i añade la información que se obtiene exclusivamente a partir de las cláusulas de Δ cuyas cabezas son átomos del estrato i, y la información del estrato inferior permanece invariable. Nótese que, si $str(A) = i$, entonces $str(\exists x_1 \ldots \exists x_n(A \approx A' \wedge G)) \leq i$. El operador T_i es monótono y continuo, es decir:

Lema 2 (Monotonía de T_i) Sea $i \geq 1$ y $I_1, I_2 \in \mathcal{I}_i$ tal que $I_1 \sqsubseteq_i I_2$. Entonces se cumple que:

$$T_i(I_1) \sqsubseteq_i T_i(I_2).$$

Lema 3 (Continuidad de T_i) Sea $i \geq 1$ y $\{I_n\}_{n\geq 0}$ una familia enumerable de interpretaciones sobre i, tal que $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$. entonces:

$$T_i(\bigsqcup_{n\geq 0} I_n) = \bigsqcup_{n\geq 0} T_i(I_n).$$

Las demostraciones de los lemas anteriores se pueden encontrar en el apéndice A de [A.3].

Al ser el operador T_i continuo para todo $i \geq 1$, por el teorema de Knaster-Tarski [118], T_1 tiene un mínimo punto fijo, denotado con fix_1, tal que:

$$fix_1 = \bigsqcup_{n\geq 0} T_1^n(I_\perp),$$

donde la interpretacion $I_\perp$ representa la función constante $\emptyset$. Procediendo de manera similar, se puede definir una cadena:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots, \sqsubseteq_i T_i^n(fix_{i-1}), \ldots,$$

para cada estrato $i > 1$,y podemos encontrar el siguiente punto fijo de dicha cadena:

$$fix_i = \bigsqcup_{n\geq 0} T_i^n(fix_{i-1}).$$

En particular, si k es el estrato máximo en Δ, fix_k se simplifica escribiendo solamente fix. Por tanto, $fix(\Delta)$ representa los pares (A, C) tales que A se puede deducir de Δ si C se satisface.

Mostramos estas nociones teóricas de manera práctica con el siguiente ejemplo.

Ejemplo 10 Dado el dominio finito $[a, b, c]$, considérese el programa Δ:

$p(a)$.
$p(b)$.

$\overline{\forall}\ t(x) \Leftarrow p(x)$.
$\overline{\forall}\ q(x) \Leftarrow \neg p(x)$.
$\overline{\forall}\ u(x) \Leftarrow \neg q(x)$.
$\overline{\forall}\ r(x) \Leftarrow (p(x) \Rightarrow q(x))$.

En este programa p y t pertenecen al primer estrato, q y r al estrato 2 y u al estrato 3. A continuación mostramos la evolución del cálculo del punto fijo para cada uno de los estratos. La relación de forzado es no determinista y, por tanto, las restricciones que se muestran en el ejemplo corresponden a representantes de sus respectivas clases de equivalencia.

- Para el primer estrato se considera el operador T_1:
  - La primera iteración corresponde a $(T_1(\emptyset))(\Delta)$. Consideramos la cláusula $p(a)$ y tratamos de probar si hay una C tal que:

    $$\emptyset, \Delta \Vdash ((x \approx a) \wedge \top), C).$$

    Una restricción válida es $C \equiv (x \approx a)$ y por tanto, el punto fijo de este primer estrato contiene el par $(p(x), x \approx a)$. Análogamente, también contiene el par $(p(x), x \approx b)$ usando la cláusula $p(b)$, por tanto:

    $$T_1(\emptyset)(\Delta) = \{(p(x), x \approx a), (p(x), x \approx b)\}.$$

    Este operador también considera la cláusula que queda del estrato 1, sin embargo, al aplicarse sobre el conjunto vacío no existen pares para esta cláusula que deban aparecer en $T_1(\emptyset)(\Delta)$.
  - La segunda iteración corresponde a $(T_1(T_1(\emptyset)))(\Delta)$.
    Utilizando ahora la cláusula restante del primer estrato, $\forall x(p(x) \Rightarrow t(x))$, se trata de probar que:

    $$T_1(\emptyset), \Delta \Vdash (\exists x'(x \approx x' \wedge p(x')), C).$$

    Para eliminar el cuantificador existencial, de acuerdo con la definición, se sustituye x' por una nueva variable y y se obtiene:

    $$T_1(\emptyset), \Delta \Vdash (x \approx y \wedge p(y)), C').$$

    De forma que $C \vdash_C \exists y C'$. Para la conjunción, siguiendo de nuevo la definición de la relación de forzado, se debe verificar $T_1(\emptyset), \Delta \Vdash (x \approx y, C')$ y $T_1(\emptyset), \Delta \Vdash (p(y), C')$. Por ejemplo, la restricción $C \equiv (x \approx a) \vee (x \approx b)$ satisface las condiciones necesarias. Por lo tanto $T_1(\emptyset)(\Delta)$ contiene el par $(t(x), x \approx a \vee x \approx b)$ que completa el punto fijo del primer estrato[2]. Así, fix_1 queda definido por el conjunto $\{(p(x), x \approx a), (p(x), x \approx b), (t(x), x \approx a \vee x \approx b)\}$. En adelante, los pasos referentes a la eliminación de $\exists$ y $\approx$ no se mostrarán para simplificar.
- Para el segundo estrato se aplica el operador T_2, que opera sobre fix_1 y sobre Δ. Por tanto, su primera y única iteración corresponde al cálculo de $(T_2(fix_1))(\Delta)$:

[2] En esta iteración se vuelven a considerar $p(a)$ y $p(b)$ pero no se pueden añadir nuevos pares a los ya existentes.

- Haciendo uso de la cláusula $\forall x(\neg p(x) \Rightarrow q(x))$, se tendrá que verificar:

$$fix_1, \Delta \Vdash (\exists x'(x \approx x' \wedge \neg p(x')), C).$$

Lo que lleva a que C debe verificar $C \vdash_{\mathcal{C}} \neg C'$ para todas las C' tales que $(p(x'), C') \in fix_1(\Delta)$. Para ello se usan los pares pertenecientes a $fix_1(\Delta)$ y se obtienen $(x \approx a)$ y $(x \approx b)$. Por lo que $T_2(fix_1(\Delta))$ contiene el par $(q(x), x \not\approx a \wedge x \not\approx b)$.

- Consideramos ahora la cláusula $\forall x((p(x) \Rightarrow q(x)) \Rightarrow r(x))$. Comprobamos si:

$$fix_1, \Delta \Vdash (\exists x'(x \approx x' \wedge p(x') \Rightarrow q(x')), C).$$

Lo cual nos lleva a aumentar Δ con $p(x')$ (según la relación de forzado) y tratar de probar si:

$$fix_1, \Delta \cup \{p(x')\} \Vdash (q(x'), C').$$

No es posible encontrar, en este caso, una C' que satisfaga esta relación y por tanto $T_2(fix_1) = T_2^j(fix_1)$ para $j \geq 1$, por lo que será un punto fijo fix_2 para el estrato 2.

- Para el tercer estrato se procede como en estratos anteriores aplicando T_3 sobre fix_2 y sobre Δ. Utilizamos la cláusula del predicado en el tercer estrato $\forall x(\neg q(x) \Rightarrow u(x))$. Procediendo de manera análoga a los anteriores pasos, el par $(u(x), x \approx a \vee x \approx b)$ aparece en $(T_3(fix_2))(\Delta)$ junto con los pares correspondientes a $fix_2(\Delta)$.

Concluimos, mostrando el punto fijo final $fix_3 = \{(p(x), x \approx a),\ (p(x), x \approx b),(t(x), x \approx a \vee x \approx b),\ (q(x), x \not\approx a \wedge x \not\approx b),(u(x), x \approx a \vee x \approx b)\}$. □

Corrección y completitud

Decir que el esquema $HH_{\neg}(\mathcal{C})$ es correcto y completo con respecto al cálculo $\mathcal{UC}_{\neg}$ es equivalente a sostener que la relación de forzado, considerando el punto fijo del último estrato de la base de datos para una una consulta concreta, coincide con la derivación en el cálculo $\mathcal{UC}_{\neg}$ para dicha consulta.

Más concretamente si $str(G) = i$ entonces fix_i fuerza (G, C) en el contexto de Δ sí y solo sí C es una restricción respuesta de G en Δ. Esta demostración aparece con todo detalle en [A.3], sin embargo en esta memoria exponemos un resumen.

Las siguiente proposición presenta un resultado de equivalencia entre la semántica de punto fijo y la semántica de pruebas para el caso sin negación que utilizamos para la demostración del resultado final.

Proposición 1 Para todo $i \geq 1$, $\Delta \in \mathcal{W}$, y para todo par $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, tal que G que no contiene negación, si $str(G) \leq i$ entonces:

$$fix_i, \Delta \Vdash (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_{\neg}} G.$$

Para el caso en que las consultas no tengan negación, la demostración de la correción y la completitud es similar a la que aparece en [35] para $HH(\mathcal{C})$. Pasamos a presentar el resultado de corrección y completitud entre la semántica de punto fijo y el cálculo $\mathcal{UC}_{\neg}$.

Teorema 1 (Corrección y completitud) Para todo $i \geq 1$, $\Delta \in \mathcal{W}$, y para todo par $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, si $str(G) \leq i$ entonces:

$$fix_i, \Delta \Vdash (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_{\neg}} G.$$

Para demostrar este resultado es necesario hacer inducción sobre i. Al tratarse de un resumen, nos centramos en el caso en el que G se corresponde con $\neg A$ dado que nos parece el más representativo.

- La proposición 1 captura el caso en que $i = 1$.
- Para el caso $i > 1$, asumimos la siguiente hipótesis de inducción: para todo Δ, G, C, con $str(G) \leq i - 1$ se cumple $fix_{i-1}, \Delta \nvDash (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G$.
  - En la proposición 1 se hace inducción sobre la estructura de G, sin considerar el caso $\neg A$.
  - Para el caso $\neg A$, partimos de que si $fix_i, \Delta \nvDash (\neg A, C) \iff$ para todo C' tal que $(A, C') \in fix_i(\Delta)$. Debemos demostrar $C \vdash_{\mathcal{C}} \neg C'$ o bien que no existe dicha C' y $C \equiv \top$. Obviamente, $str(A) \leq i - 1$, y lo anterior es equivalente a decir que para toda restricción C' tal que se cumpla $fix_{i-i}, \Delta \nvDash (A, C')$, debemos demostrar $C \vdash_{\mathcal{C}} \neg C'$ o bien que no existe dicha C' y $C \equiv \top$.

    Aplicar la hipótesis de inducción es equivalente a decir que: o bien por cada C' tal que $\Delta; C' \vdash_{\mathcal{UC}_\neg} A$, demostramos $C \vdash_{\mathcal{C}} \neg C'$ o bien que no hay tal C' y $C \equiv \top$. Esto equivale a la definición de la regla del cálculo $\Delta; C \vdash_{\mathcal{UC}_\neg} \neg A$, como se puede ver en la definición 2.5:

$$\frac{\Gamma \vdash_{\mathcal{C}} \neg C \quad \text{para todo } \Delta; C \vdash A}{\Delta; \Gamma \vdash \neg A} (\neg)$$

Todos los análisis de casos mencionados aparecen en el apéndice A de [A.3]. Finalmente, como consecuencia del teorema extraemos que:

$$(A, C) \in fix(\Delta) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} A.$$

Lo que significa que, tal y como queríamos demostrar, los átomos del punto fijo de la base de datos son los que se derivan del cálculo utilizando la misma restricción en cada caso.

La ventaja de nuestra semántica de punto fijo es que se puede usar como base para la implementación del sistema $HH_\neg(\mathcal{C})$ que presentamos a continuación. Para estos formalismos usamos un sistema de restricciones genérico $\mathcal{C}$. Para el sistema podemos ver a $\mathcal{C}$ como una caja negra capaz de comprobar la $\mathcal{C}$-satisfactibilidad de una restricción de entrada C.

2.3. El sistema $HH_\neg(\mathcal{C})$

En las secciones anteriores hemos presentado $HH_\neg(\mathcal{C})$ como lenguaje formal, así como dos semánticas. En esta sección presentamos una implementación de un sistema de bases de datos deductivo con restricciones basado en dicho lenguaje.

En la implementación del sistema podemos distinguir dos bloques. El primero corresponde a la implementación de la semántica de punto fijo. Como hemos señalado, esta implementación está guiada por la definición de la semántica y es independiente del sistema de restricciones (véase la sección 2.3.6). La otra parte corresponde a la implementación del sistema de restricciones (véase la sección 2.3.3).

En esta sección se recopilan los contenidos aparecidos en [A.1] respecto a la implementación del núcleo del sistema y las restricciones de integridad introducidas en [A.3].

El sistema está disponible en la dirección

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`

junto con una batería de ejemplos y un manual de usuario.

En los ejemplos del sistema usamos una sintaxis concreta para las cláusulas bastante cercana a la sintaxis de Prolog, donde los predicados y los símbolos de constantes comienzan con minúscula y las variables comienzan con mayúscula. Utilizamos "`,`"para la conjunción $\wedge$ y "`;`"para la disyunción $\vee$. Además usamos `not` para la negación, `D=>G` para la implicación $D \Rightarrow G$, `ex(X,G)` representa $\exists x\ G$, `fa(X,G)` se usa para $\forall x\ G$ y `constr(Dom,C)` para una restricción C junto con su dominio Dom. El sistema también requiere declaración explícita de tipos para los predicados mediante:

$$\texttt{type}(\textit{predicate}(\textit{dom\_1, ..., dom\_n})).$$

En general se pueden usar distintos resolutores para la misma base de datos; sin embargo, no se pueden combinar en una misma restricción. Es decir, no puede haber relaciones que combinen, por ejemplo, dominio finito y dominio real en la parte intensional de la base de datos, como $\texttt{interestRate}(\texttt{I}, 2{,}0)$ `:-` $\texttt{client}(\texttt{I}, \texttt{B}, \texttt{S})$, $\texttt{constr}(\texttt{real}, \texttt{B} < 1200{,}0)$ donde `I` es una variable de dominio finito y `B` es una variable de dominio real.

Por tanto, los predicados con argumentos de distintos dominios son solamente los definidos extensionalmente y se usarán para que la base de datos sea más legible. A continuación, presentamos la declaración explícita de los dominios enumerados del ejemplo 6:

```
domain(client_dt,[smith,brown,mcandrew]).
domain(branch_dt,[lon,ny,mad,par]).
```

Para evitar la combinación de dominios durante el cómputo en la base de datos definimos la relación `client_id` que asocia un identificador a cada cliente:

```
client_id(smith,1.0)
client_id(brown,2.0)
client_id(mcandrew,3.0).
```

El resto de predicados extensionales del ejemplo 6 se definen igual, salvo que cambiamos el nombre de los clientes por su identificador. Es decir, escribimos `interestRate(I,2.0)` en lugar de `interestRate(I,brown)`.

Añadimos además un ejemplo de uno de los predicados intensionales para mostrar la sintaxis que procesa el sistema.

```
interestRate(I,2.0):- client(I,B,S), constr(real,B<1200.0).
interestRate(I,5.0):- client(I,B,S), constr(real,B>=1200.0).
```

2.3.1. Fases de cómputo

A continuación esquematizamos las distintas fases de cómputo del sistema al calcular el punto fijo de una base de datos `Delta`. Los distintos módulos del sistema aparecen en la figura 2.7.

Al calcular el punto fijo de una base de datos `Delta`, el sistema:

1. Comprueba e infiere los tipos de los predicados.
2. Construye el grafo de dependencias para `Delta`.

3. Calcula la estratificación s para **Delta**, si existe. Si no existe el sistema lanza un mensaje de error y se detiene.

4. Si la fase anterior tiene éxito, calcula fix(**Delta**) (véase la sección 2.3.6).

El sistema mantiene en memoria el punto fijo fix(**Delta**), la estratificación s y el grafo de dependencias para **Delta**.

Para la implementación de implicaciones anidadas y los agregados en el sistema hemos aprovechado la noción de grafo de dependencias para asegurar un cálculo correcto. Además de las dependencias que se explican en la sección 2.2.2, hemos añadido otras nuevas dependencias negativas para tratar las funciones de agregación (véase la sección 2.3.4) y las implicaciones anidadas (véase la sección 2.3.7).

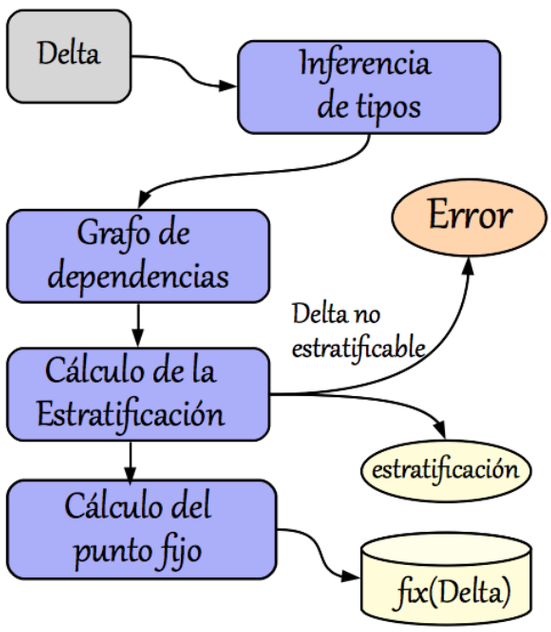


Figura 2.7: Fases del sistema

Pasamos a presentar las distintas fases de cómputo de nuestro sistema con la base de datos del ejemplo 6. Como hemos dicho, en primer lugar se infieren los tipos de los predicados y se calcula la estratificación. Para los predicados del ejemplo 6 la estratificación calculada es:

$$s = [(\texttt{client}, 1), (\texttt{pastDue}, 1), (\texttt{mortgageQuote}, 1), (\texttt{debtor}, 1), (\texttt{interestRate}, 1), (\texttt{hasMortgage}, 1), (\texttt{accounting}, 1), (\texttt{client\_id}, 1), (\texttt{branch}, 1), (\texttt{newMortgage}, 2), (\texttt{gotMortgage}, 2), (\texttt{personalCredit}, 3)].$$

A continuación mostramos el punto fijo fix(**Delta**) de la base de datos propuesta que debe contener los pares (compuestos por un átomo y una restricción asociada) que corresponden a la parte extensional de la base de datos, por ejemplo:

$$(\texttt{client}(3{,}0, 5300{,}0, 3000{,}0), \texttt{true}),$$

así como los pares que se corresponden con la parte intensional:

- En el estrato 1:

```
(debtor(1.0), true),
(interestRate(2.0,2.0), true),
(interestRate(X,Y),
 ((X=1.0, Y=5.0); (X=3.0, Y=5.0))),
(accounting(X,Y,Z),
 ((Y=400.0, Z=1500.0, X=2.0); (Y=100.0, Z=3000.0, X=3.0))),
(hasMortgage(X), (X=2.0;X=3.0))
```

- En el estrato 2:

```
(newMortgage(X,Y),
 ((Y=<200.0, X=2.0); (Y=<1100.0, X=3.0))),
(gotMortgage(X), (X=2.0; X=3.0))
```

- En el estrato 3:

```
(personalCredit(X,Y),
 ((Y>=6000.0, Y<20000.0, X/=2.0, X/=3.0);
(Y<6000.0, X=2.0); (Y<6000.0, X=3.0)))
```

2.3.2. Consultas

Cuando se formula una consulta G al sistema, este calcula, si existe, una nueva estratificación s' para el conjunto Delta $\cup$ {G}. Si no existe dicha s' la consulta no puede ser calculada. Si por el contrario, el sistema logra calcular la nueva s', usamos el punto fijo almacenado para calcular la respuesta.

Tal y como explicamos en la sección 2.2, la respuesta C debe satisfacer

$$fix(\texttt{Delta});\ \texttt{Delta} \Vdash (\texttt{G}, \texttt{C}).$$

Esta relación de forzado se implementa haciendo uso del predicado:

$$\texttt{force}(\texttt{Delta}, \texttt{Stratification}, \texttt{I}, \texttt{G}, \texttt{C})$$

que explicamos en la sección 2.3.6. Este predicado usa la estratificación actual s. Para el cómputo del resultado de una consulta distinguimos dos casos:

- Si $s = s'$, entonces como fix(Delta) se calcula con s, la restricción respuesta C se puede obtener ejecutando force(Delta,s,fix(Delta),G,C).
- Si $s \neq s'$, es porque G contiene alguna subconsulta de la forma D=>G'.
  - En este caso el grafo de dependencias de Delta $\cup$ {G} (del cual se ha obtenido s') contiene los arcos correspondientes a Delta además de aquellos correspondientes a las implicaciones de G.
  - La nueva estratificación s' es válida también para Delta y con ella se obtiene el mismo punto fijo fix(Delta).
  - Por tanto, al igual que sucede en el caso anterior, la restricción respuesta C se obtiene ejecutando el predicado force con la nueva estratificación:

force(Delta,s',fix(Delta),G,C).

Necesitamos la información de la nueva estratificación s' porque al resolver un objetivo G, debemos tener en cuenta también las implicaciones anidadas dentro de ella. Cuando vamos a resolver D=>G', de acuerdo con la definición de la relación de forzado, aumentamos localmente nuestra base de datos Delta con la cláusula D dado que la respuesta a la consulta G depende de fix(Delta $\cup$ {D}).

En conclusión, puesto que la estratificación s' se ha definido teniendo en cuenta dichas implicaciones, es también una estratificación válida para Delta $\cup$ {D}. Por tanto podemos asegurar un cómputo correcto de la consulta en ambas situaciones.

Siguiendo con el ejemplo del banco, mostramos el cómputo para diferentes consultas incluyendo casos en los que la estratificación inicial es válida y casos en los que no. Como primer ejemplo preguntamos si todo cliente pertenece a la oficina de Madrid:

```
hhnc> fa(A,branch(mad,A)).
```

Esta consulta no requiere ningún cambio en el grafo de dependencias y se puede resolver usando el punto fijo almacenado. Una cuantificación universal sobre un dominio finito se convierte en una restricción conjuntiva. Dicha restricción se resuelve instanciando la variable con cada elemento del dominio. El resultado es trivialmente:

```
Answer: false
```

Como ejemplo de negación, podemos preguntar los clientes que no tienen hipoteca:

```
hhnc> not(hasMortgage(N)).
```

Esta consulta tampoco cambia la estratificación y el sistema utiliza de nuevo el punto fijo almacenado. Para obtener la respuesta a esta consulta el sistema busca en el punto fijo todas las restricciones C de la forma (hasMortgage(N),C) y después crea una conjunción de negaciones de las restricciones (N=3.0 y N=2.0 presentes en el punto fijo para hasMortgage) que se envía al resolutor. Finalmente la respuesta obtenida es:

```
Answer: N/=3.0, N/=2.0
```

Por último presentamos un ejemplo de consulta que fuerza al cambio de la estratificación. Este ejemplo no tiene un significado natural, solo trata de ilustrar esta situación:

```
hhnc> newMortgage(N,R) => interestRate(N,R).
```

La consulta introduce una nueva dependencia entre newMortgage e interestRate, por tanto, interestRate pasa a estar en el estrato 2 en la nueva estratificación s'. Sin embargo, como hemos explicado antes, el punto fijo almacenado es válido para calcular esta consulta, que tiene por resultado:

```
Answer: (R=2.0, N=2.0); (R=5.0, N=1.0); (R=5.0, N=3.0)
```

De está forma, aunque la consulta demande cambiar la estratificación, podemos usar el punto fijo almacenado.

2.3.3. Implementación de los resolutores

Para nuestro sistema implementamos tres sistemas de restricciones como posibles instancias del esquema $HH_{\neg}(\mathcal{C})$: booleanos, reales y dominios finitos.

Los sistemas de restricciones usan la siguiente notación concreta para cierto, falso, funciones booleanas y cuantificadores:

$$\texttt{true},\quad \texttt{false},\quad \texttt{not},\quad \texttt{ex(X,C)}.$$

Al igual que Prolog usamos "`;`"para la disyunción y "`,`"para la conjunción. También incluimos entre otros los siguientes operadores de comparación:

$$=,\quad /=,\quad >,\quad >=.$$

que tienen el significado habitual.

Las restricciones numéricas incluyen operadores arítmeticos (como `+`, `-`, ...) y símbolos de función (como por ejemplo `abs` para el valor absoluto). Además, los booleanos y los dominios finitos admiten el cuantificador universal (`fa(X,C)`).

Para los dominios finitos proporcionamos una restricción de dominio "`X in Range`", donde `Range` es un subrango de valores que se construye con `V1..V2`, que denota el conjunto de valores comprendidos en el intervalo cerrado entre `V1` y `V2`, y `R1\/R2`, que representa la unión de rangos.

Para la implementación de los sistemas de restricciones, i.e., para implementar $\vdash_{\mathcal{C}}$, se ha tomado como punto de partida la relación de deducibilidad de la lógica clásica con igualdad. Esta relación de deducibilidad satisface los requisitos mínimos que imponemos a los sistemas de restricciones en el fundamento teórico (véase la sección 2.1).

Como esquematizamos en la figura 2.8, para la implementación de esta relación hemos desarrollado una interfaz genérica `solve(I,C,SC)` para la relación $\mathtt{C} \vdash_{\mathcal{C}} \mathtt{SC}$. En general `solve` genera o bien una *restricción respuesta* `SC` a partir de `C`, si es satisfactible, o `false` en otro caso.

Para manejar consultas con agregados hemos tenido que añadir a la interfaz `solve` la interpretación I sobre la que se calcula las funciones de agregado.

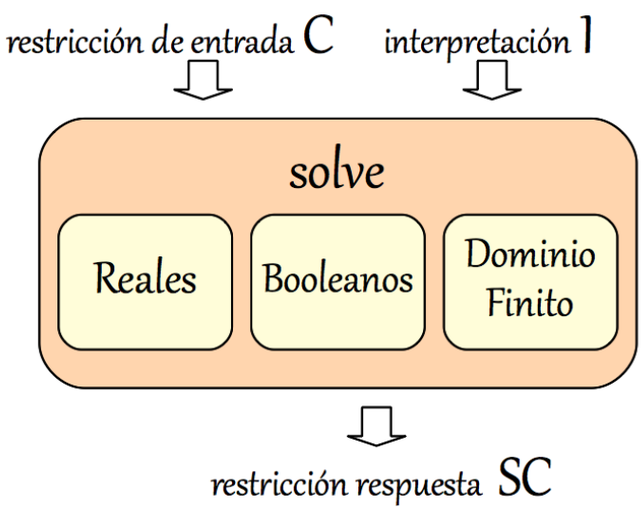


Figura 2.8: Interfaz genérica de los resolutores del sistema.

SC es también una restricción del sistema de restricciones $\mathcal{C}$, pero simplificada y más legible, que puede ser una *restricción simple* o bien una disyunción de restricciones simples. Llamamos *restricción simple* a aquella que no contiene ni disyunciones, ni cuantificadores ni negaciones.

La interfaz genérica `solve` se implementa con el predicado:

```
solve(+Interpretation,+Constraint,-SolvedConstraint)
```

que acepta como entrada una restricción `Constraint` y devuelve una restricción respuesta `SolvedConstraint` bajo una `Interpretation`. Para la implementación de los resolutores hemos utilizado los de SWI-Prolog [129, 119]. Los resolutores de $HH_{\neg}(\mathcal{C})$ están implementados como una capa superior sobre estos resolutores de SWI-Prolog porque, en general, las restricciones de nuestro sistema son más complejas que las que admiten los resolutores subyacentes. Por tanto, la idea tras la interfaz `solve` es:

1. Discriminar el tipo de restricción para decidir a qué resolutor (reales, dominios finitos o booleanos) la envía.
2. Transformar dicha restricción en una restricción más sencilla y válida como entrada de los resolutores de SWI-Prolog. Las restricciones que los resolutores subyacentes resuelven se consideran restricciones primitivas.
3. Enviarla a los resolutores subyacentes para obtener una respuesta.
4. Finalmente, recomponer el resultado.

Como ejemplo de la implementación de los resolutores, consideramos el sistema de restricciones de los dominios finitos $\mathcal{FD}$. Para esta instancia, nuestros resolutores asocian a los valores (no enteros en general) otros valores numéricos enteros del sistema subyacente, i.e., antes de enviar al resolutor una restricción, se reescribe usando el valor entero correspondiente y tras la resolución se hace el proceso inverso.

Para restricciones más complejas (cuantificaciones y disyunciones) que los resolutores SWI-Prolog no manejan, se han implementado resolutores específicos (veánse los apéndices C.2 y C.3 de [A.3]).

2.3.4. Funciones de agregación

Las funciones de agregación se utilizan habitualmente en los sistemas de bases de datos relacionales para obtener valores de resumen partiendo de conjuntos de valores no necesariamente numéricos. En esta sección explicamos cómo extendemos nuestro sistema de bases de datos deductivo con restricciones para tratar con agregados en el contexto del sistema de restricciones.

Cuando tratamos de añadir funciones de agregación al lenguaje de restricciones de nuestro sistema de bases de datos, debemos tener en cuenta algunas características. Las funciones de agregación toman un conjunto de valores no necesariamente número y devuelven un valor único. Además, dado que la restricción respuesta puede representar un conjunto infinito, algunos agregados (por ejemplo el recuento `count`) pueden no tener sentido. Para resolver estos obstáculos hemos aprovechado ciertos aspectos de la semántica de punto fijo de $HH_{\neg}(\mathcal{C})$.

- Por un lado, la semántica de punto fijo estratificado que ha sido diseñada para tratar la negación ha resultado un marco adecuado para incorporar funciones de agregación. Dado que debemos garantizar la monotonía del cómputo, cuando se calcula el punto fijo para

un estrato dado, el resultado de una función de agregación sobre este estrato no debe cambiar dinámicamente.

Por ejemplo, el recuento de tuplas para un estrato se puede averiguar cuando ese estrato se ha calculado totalmente. Para asegurarnos de que este cómputo ha finalizado, antes de calcular la función de agregación, nos interesa que la relación a la que hace referencia la función de agregación esté en un estrato inferior, para lo que se añadirán nuevas dependencias al grafo, de forma análoga a cómo manejamos la negación en el sistema.

- Los agregados se pueden representar como funciones del lenguaje de restricciones y, por tanto, para implementar las funciones de agregación hemos delegado su cómputo al correspondiente resolutor de restricciones.
- Por otro lado, para calcular el resultado de una función de agregación sobre un predicado p para cada par $(p(x_1, \ldots, x_n), C)$ en la interpretación actual, se debe cumplir que C restringe cada variable $x_1, \ldots, x_n$ a un valor concreto. Es decir, no podemos calcular sumatorio de los valores de variable `X` en el predicado `p` si tenemos el par `(p(X),X>3)` dado que su valor sería infinito. En el caso contrario, nuestro sistema no es capaz de obtener un resultado para esta función.

A continuación presentamos mediante unos ejemplos las funciones de agregación en nuestro sistema. Hemos implementado las funciones **count** (recuento) sobre un predicado, también **sum** (sumatorio), **avg** (media), **min** (mínimo) y **max** (máximo) sobre las variables de un predicado. Las funciones de agregación aparecen dentro de una restricción del sistema:

```
constr(dom,res_agg).
```

Donde **dom** representa el dominio de dicha restricción y **res_agg** representa una restricción que puede incluir funciones de agregado. Un ejemplo concreto de **res_agg** es:

$$\texttt{X} = \texttt{min}(\texttt{p}(\texttt{A}, \texttt{B}, \texttt{C}), \texttt{B}))$$

para calcular la función de agregado mínimo sobre el predicado `p`. El resultado es una restricción `X = val` donde `val` es el mínimo de los valores que toma la variable `B` para todos los pares asociados al predicado `p` en el punto fijo de la base de datos. La única que presenta una notación distinta es la función de recuento que tiene como único argumento el predicado al que se aplica.

Los detalles de la implementación se pueden encontrar en el apéndice C.2 de [A.3]. Pasamos a presentar algunos ejemplos prácticos.

Ejemplo 11 Por ejemplo, la vista:

```
liquid(Amount) :- constr(real,Amount=sum(client(N,B,S),B)).
```

definida para el ejemplo 6 de la base de datos para un banco, permite calcular la suma de los balances de los clientes, incluyendo la función de agregación **sum** en la restricción. Otro ejemplo de uso de las funciones de agregación **sum** y **count** para definir el salario medio es:

```
avg_salary(Average) :- constr(real,Average=
                              sum(client(N1,B1,S1),B1)/
                              count(client(N2,B2,S2)).
```

La riqueza del lenguaje de bases de datos $HH_{\neg}(C)$ permite calcular una función de agregación combinada con una asunción que cambia los valores de la agregación. Por ejemplo, se puede escribir la siguiente vista que contiene la hipótesis de que el cliente Brown tiene una deuda,

```
view(X) :- pastDue(brown,200.0) =>
                  constr(real,X=sum(pastDue(N,A),A))
```

Esta vista devolverá la suma de todas las deudas de los clientes en la base de datos de ejemplo supuesta una nueva deuda para Brown. Añadiendo esta vista a la base de datos, podemos calcular el resultado:

```
HHn(C)> view(X).
```

que tiene como respuesta `X=3300`. □

De igual forma a lo que ocurre con una cláusula con un átomo negado en su cuerpo, si una de las cláusulas que definen un predicado p contiene un agregado sobre el predicado q, el cómputo de q debe haber finalizado antes del comienzo del cómputo de p. Esta condición se puede conseguir fácilmente introduciendo una dependencia negativa desde q hacia p, lo que garantiza que el estrato de q es menor que el estrato de p, es decir en la estratificación se cumple que $s(q) < s(p)$. Para los ejemplos anteriormente propuestos se debe cumplir que:

$$s(\texttt{client}) < s(\texttt{liquid}),\ s(\texttt{client}) < s(\texttt{avg\_salary}),$$

y también que:

$$s(\texttt{pastDue}) < s(\texttt{view}).$$

2.3.5. Restricciones de integridad

El término integridad de datos se refiere a la corrección de los datos en una base de datos con respecto a las restricciones de integridad especificadas por el usuario. La integridad de los datos almacenados puede perderse de muchas maneras diferentes. Por ejemplo, pueden añadirse datos no válidos a la base de datos tales como un pedido que especifica un producto no existente. Estas restricciones deben ser definidas por la persona que modela la base de datos.

En esta sección presentamos nuestra propuesta de implementación de restricciones de integridad [A.2] para el sistema $HH_{\neg}(C)$. La idea tras esta propuesta es:

- En primer lugar, el usuario define un predicado utilizando el lenguaje $HH_{\neg}(C)$ para especificar la restricción de integridad.
- Durante el cómputo del punto fijo, se le asocia una restricción a dicho predicado.
  - Si la restricción se satisface y se simplifica a `true`, el cómputo termina con normalidad.
  - Si por el contrario, se simplifica a `false` la restricción de integridad se considera no cumplida, lo que conlleva que el cómputo se detenga y se muestre un mensaje de error.

En general, al implementar restricciones de integridad se deben tener en cuenta varios aspectos sobre la implementación del sistema que detallamos a continuación.

Declaración de las restricciones de integridad

Como primer ejemplo de las restricciones de integridad volvemos a extender el ejemplo 6. La restricción de clave primaria (que evitaría que se introduzcan dos clientes con el mismo identificador) se puede especificar definiendo una relación en la base de datos para la que su restricción respuesta es *true*, si no se cumple la condición de que todos los clientes tengan diferente identificador. En concreto añadiendo la siguiente cláusula a la base de datos:

```
pk_id_fails:- client_id(A,B), client_id(A,D),constr(real,(B/=D)).
```

Este ejemplo nos da una idea de que el problema de las restricciones de integridad se puede representar de forma sencilla en el lenguaje $HH_{\neg}(\mathcal{C})$ dado que incluye restricciones.

Incumplimiento de una restricción de integridad

Nuestro sistema implementa un cómputo de punto fijo iterativo que va añadiendo tuplas en cada interpretación. Por tanto, un momento seguro para detectar que una restricción no se cumple es cuando se añaden las tuplas. Nótese que, al permitir implicaciones, una restricción puede no cumplirse también durante un cómputo local (necesario para una consulta hipotética) que luego será posteriormente descartado. En el siguiente ejemplo vemos cómo es posible que una restricción de integridad no se cumpla en un cómputo local.

Ejemplo 12 Partimos del siguiente programa

```
p(0,0).
q :- p(0,1) =>  r.
```

Si suponemos el predicado `pk_p_fails` como

```
 pk_p_fails:- p(A,B), p(C,D), constr(data,(A/=C)).
```

en el cómputo local en el que se añade `p(0,1)` esta restricción de integridad se estaría incumpliendo. Sin embargo, si se espera al final del cómputo para la comprobación, esta violación nunca sería detectada. □

Por tanto, como acabamos de indicar una aproximación segura al problema es hacer la comprobación cada vez que se añaden tuplas a la interpretación.

Respuesta al incumplimiento de una restricción de integridad

Lo habitual para sistemas de bases de datos es el lanzamiento de una excepción que indique la violación de la restricción de integridad. En nuestro sistema en cuanto se detecta la violación de una restricción de integridad se detiene el cómputo y se lanza un mensaje de error.

Implementación de clave primaria

En esta sección nos centramos en el manejo de la claves primarias aunque nuestro sistema también permite restricciones de *clave ajena* y *dependencias funcionales* que se tratan de forma similar (véase la sección 3 de [A.2]).

Cuando un usuario define una base de datos y quiere incluir una restricción de integridad de clave primaria sobre un subconjunto de argumentos del predicado **pred** debe añadir además un predicado adicional de la forma **pk(pred**($\mathtt{X_1}$...$\mathtt{X_N}$), ($\mathtt{X_i}$...$\mathtt{X_j}$)), donde ($\mathtt{X_i}$...$\mathtt{X_j}$) es el subconjunto de variables de ($\mathtt{X_1}$, ..., $\mathtt{X_N}$) que conforman la clave primaria.

Volviendo al ejemplo 6, utilizando un aserto de Prolog introducimos el predicado que se debe añadir a la base de datos para especificar la restricción de clave primaria sobre una relación que asigna una contraseña a cada identificador de cliente.

```
:- pk(client_pwd(Id,Pwd),Id).
```

Que significa que **Id** es la clave primaria de **client_pwd**. Por tanto, si ya tenemos en la base de datos:

client_pwd(1,123)

esta restricción impide que haya otra contraseña asignada al identificador **1**.

Como hemos señalado, el sistema delega en el resolutor de restricciones el cómputo de las restricciones de integridad. Calcula una restricción respuesta que será cierta si se cumple la restricción y falsa en caso contrario. En caso de de que la restricción sea falsa el cómputo se detiene y se muestra el mensaje de error. Todos los detalles de la implementación de las restricciones de integridad de nuestro sistema, así como un ejemplo paso a paso se pueden encontrar en [A.2].

2.3.6. Cómputo de la semántica de punto fijo

La implementación del sistema se basa en la semántica de punto fijo que hemos presentando en la sección 2.2.2. Sin embargo, debido a la presencia de implicaciones anidadas, hay determinados aspectos de la implementación que requieren una explicación adicional. En esta sección se mostrará una visión general de dicha implementación presentando cómo se ha tratado la relación de forzado para el caso de la implicación. En adelante los resolutores de restricciones serán invocados como cajas negras mediante la llamada a su interfaz genérica **solve**.

Asumimos que Δ es una base de datos estratificable cuya estratificación s ha sido calculada en anteriores fases del cómputo. Como ya hemos señalado tenemos almacenada Δ junto con la estratificación s (que representamos como una lista **Stratification**). El punto fijo se calcula estrato por estrato, aunque en ocasiones, algún estrato deberá ser recalculado de manera local debido a la presencia de implicaciones anidadas. Este caso se explica con detalle en la sección 2.3.7.

Para comenzar, el predicado:

fixPointStrat(+Delta, +Stratification, +St, -Fix)

calcula **Fix** $= fix_{\mathtt{St}}$(**Delta**) haciendo uso de **Stratification**, siendo **Delta** una base de datos tal que str(**Delta**) $= k$. Este predicado devuelve fix_k(**Delta**) mediante el cómputo sucesivo de los puntos fijos de los estratos anteriores que van desde **St** $= 1$ hasta **St** $= k$.

Cada punto fijo se calcula iterando el operador que hemos introducido en la definición 7. Como hemos señalado en secciones anteriores, la implementación de la relación de forzado se corresponde con el predicado:

force(+Delta,+Stratification,+I,+G,-C)

Recordemos que la relación de forzado para el cálculo del punto fijo utiliza una interpretación $\mathtt{I} = T_i^n(fix_{i-1})(\mathtt{Delta})$, para algún $n \geq 0$ y un determinado estrato $i > 0$. La llamada a **force** tendrá éxito si se cumple que:

$$T_i^n(fix_{i-1}),\ \mathtt{Delta} \Vdash (\mathtt{G}, \mathtt{C}).$$

El predicado **force** se implementa de manera determinista, haciendo una distinción de casos sobre la sintaxis del objetivo **G**. Todo el código de la implementación de la relación de forzado aparece en el apéndice D de [A.3].

Finalmente nótese que el predicado **force(Delta,Stratification,G,C)** debe construir una restricción respuesta **C** tal que **I**, **Delta** $\Vdash$ **(G,C)**. Esta restricción respuesta puede ser o bien una restricción simple o una disyunción de restricciones como hemos visto en la sección anterior.

Todos los detalles de la implementación tanto de la relación de forzado, como el código Prolog se explican en [A.1]. Además, encontramos todo el código de la implementación de **force** en el apéndice D de [A.3].

2.3.7. El caso de la implicación

La implementación de **force(Delta,I,(D=>G),C)** requiere una explicación adicional que sigue la presentación que aparece en la sección 8.1 de [A.3]. A continuación presentamos el código Prolog que implementa el forzado de la implicación en el sistema.

```
force(Delta,Stratification,I,(D=>G),C) :- !,
    elab(D,De),
    localClauses(De,Ls),
    addLocalClauses(Ls,Delta,Delta1),
    getMaxStrat(G,Stratification,StG),
    fixPointStrat(Delta1,Stratification,StG,Fix),
    force(Delta1,Stratification,Fix,G,C).
```

Los predicados prolog: **elab**, **localClauses** y **addLocalClauses** se utilizan para transformar las cláusulas del programa en una representación interna que maneja el sistema y aparecen explicados en detalle en el apéndice D de [A.1].

Siguiendo la definición de la relación de forzado (véase la definición 6), la base de datos **Delta** se aumenta con la cláusula **D** (la hipótesis). Hasta el momento, la interpretación **I** se ha calculado con respecto a la base de datos original **Delta**. Si consideramos el estrato i y la iteración n entonces $(A, C) \in \mathtt{I}$ significa $(A, C) \in T_i^n(I')(\mathtt{Delta})$, donde I' es el punto fijo del estrato $i - 1$ construido para **Delta**. Según la teoría, el siguiente paso sería probar:

$$T_i^n(I'), \mathtt{Delta} \cup \{\mathtt{D}\} \Vdash (\mathtt{G}, \mathtt{C}).$$

Pero el problema es ¿cómo calculamos $T_i^n(I')(\mathtt{Delta} \cup \{\mathtt{D}\})$? En este caso nuestra interpretación **I** no es válida por dos motivos:

- En primer lugar, la relación $I(\Delta) \subseteq I(\Delta \cup \{D\})$ no se satisface para todo I, Δ y D.
- En segundo lugar, porque **I** se ha construido para la base de datos **Delta**. En concreto, el punto fijo I' se ha calculado para **Delta**, y representa el punto fijo $fix_{i-1}(\mathtt{Delta})$.

Por lo tanto, no tenemos la información necesaria del conjunto buscado, i.e., la información para la base de datos extendida:

$$\mathcal{T}_i^n(I')(\texttt{Delta} \cup \{\texttt{D}\}).$$

El problema descrito es que el operador de punto fijo $\mathcal{T}_i$ no es *constructivo* para el caso de la implicación debido al aumento del conjunto de cláusulas. Para resolver este problema se han impuesto unas restricciones sintácticas que garanticen que el cómputo sea correcto y terminante, de nuevo haciendo uso de la estratificación y del grafo de dependencias como explicamos a continuación.

Sea $\texttt{StG} = max\{St \mid (p, St) \in \texttt{Stratification}$, p un símbolo de predicado en $\texttt{G}\}$. El punto fijo del estrato $\texttt{StG}$ para $\texttt{Delta} \cup \{\texttt{D}\}$ se calcula localmente y debemos probar que:

$$fix_{\texttt{StG}}, \texttt{Delta} \cup \{\texttt{D}\} \Vdash (\texttt{G}, \texttt{C}).$$

Consideramos ahora una cláusula en $\texttt{Delta}$ de la forma $\texttt{A:-D=>G}$, tal que $i = str(\texttt{A})$. Partiendo de la definición 3, podemos deducir $\texttt{StG} \leq i$. Durante el cálculo de $fix_i(\texttt{Delta})$, el par $\texttt{(A,C)}$ se añadirá a la interpretación $\texttt{I}$ y, siguiendo la definición de la relación de forzado, debemos probar que:

$$\texttt{I, Delta} \Vdash (\exists\overline{x}(\texttt{A} \approx \texttt{A'} \wedge \texttt{D=>G}), \texttt{C})$$

Tras la eliminación de los cuantificadores existenciales, ejecutamos en primer lugar:

$$\texttt{force(Delta,Stratification,I,A} \approx \texttt{A',C)}$$

y después la llamada a:

$$\texttt{force(Delta,Stratification,I,(D=>G),C)}.$$

Este segundo $\texttt{force}$ llamará a su vez a:

$$\texttt{fixPointStrat}(\texttt{Delta1}, \texttt{Stratification}, \texttt{StG}, \texttt{Fix}),$$

donde $\texttt{Delta1} = \texttt{Delta} \cup \{\texttt{D}\}$. Hasta aquí el cómputo es correcto. Sin embargo:

- Si $\texttt{StG} = i$. Al tratar de construir $fix_i(\texttt{Delta1})$ la cláusula $\texttt{A:-D=>G}$ debe ser probada de nuevo, dado que el estrato de $\texttt{A}$ es i. Este cómputo nos lleva a un bucle no terminante porque se ejecuta $\texttt{force(Delta1,Stratification,I,(D=>G),C)}$ y $\texttt{Delta1}$ aumenta con la elaboración de $\texttt{D}$ una vez más, obteniendo $\texttt{Delta2}$, que se aumenta de nuevo con $\texttt{D}$ y así de forma no terminante.
- En caso de que $\texttt{StG} < i$, entonces $\texttt{Fix} = fix_{\texttt{StG}}(\texttt{Delta1})$ se puede construir correctamente, puesto que al ser $str(\texttt{A}) = i$, la cláusula $\texttt{A:-D=>G}$ no se considera en los estratos menores que i.

En conclusión, necesitamos garantizar que $\texttt{StG} < str(\texttt{A})$. Para asegurar que se cumpla esta condición, el predicado con mayor estrato en $\texttt{G}$ deberá depender negativamente del símbolo de predicado de $\texttt{A}$ y, por tanto, para imponer esta condición añadimos al grafo de dependencias arcos etiquetados negativamente desde todo símbolo de predicado definido que aparece en $\texttt{G}$ hacia el símbolo de predicado de $\texttt{A}$.

Esta condición adicional añade más restricciones sintácticas a la hora de que una base de datos sea estratificable. Sin embargo, mantiene la implementación correcta y completa con

respecto al marco teórico. Esto supone una pérdida de expresividad dado que habrá más bases de datos no estratificables. Sin embargo, en la práctica, no es fácil encontrar ejemplos de bases de datos reales que no cumplan esta nueva restricción sintáctica.

En el siguiente ejemplo, extraído de la sección 8.2 de [A.3], presentamos el funcionamiento de la implicación para una base de datos concreta.

Ejemplo 13 Consideramos la base de datos `Delta`:

```
q(a).
r(c).
q(b).
p(X) :- q(X) => r(X).
```

Como hemos explicado, si consideramos una estratificación donde todos los predicados `p`, `q` y `r` pertenecen al estrato 1, tendríamos una secuencia infinita de llamadas

$$\texttt{fixPointStrat}(\texttt{Delta}, \texttt{Stratification}, 1, \texttt{Fix})$$
$$\texttt{fixPointStrat}(\texttt{Delta} \cup \{\texttt{q(X)}\}, \texttt{Stratification}, 1, \texttt{Fix})$$
$$\texttt{fixPointStrat}(\texttt{Delta} \cup \{\texttt{q(X)}\} \cup \{\texttt{q(X)}\}, \texttt{Stratification}, 1, \texttt{Fix})$$
$$\ldots$$

Sin embargo, con la nueva definición de grafo de dependencias, la `Stratification` obliga a que `p` pertenezca al estrato 2, mientras que `q` y `r` permanecen en el estrato 1.

Para el primer estrato:

$$\texttt{fixPointStrat}(\texttt{Delta}, \texttt{Stratification}, 1, \texttt{Fix1})$$

obtiene `Fix1={(q(X),X=a),(q(X),X=b),(r(X),X=c)}`, porque `p(X) :- q(X)=>r(X)` ya no se considera en este estrato. Para el segundo y último estrato, en la llamada:

$$\texttt{fixPointStrat(Delta,Stratification,2,Fix2)}$$

`Fix2` se construirá a partir `Fix1` añadiendo los pares de la relación `p`. En la primera iteración, la cláusula que define `p` requiere ejecutar:

$$\texttt{force(Delta,Stratification,Fix1,q(X)=>r(X),C)},$$

para calcular la restricción `C` y poder añadir `(p(X), C)` en `Fix1`. Distinguimos dos pasos en la ejecución

1. La base de datos `Delta` se extiende con `q(X)`, para obtener `Delta1` y se evalúa localmente el punto fijo del estrato 1 (el estrato de `r`) para la base de datos extendida `Delta1`. Esto se consigue mediante la llamada:

   $$\texttt{fixPointStrat(Delta1,Stratification,1,Fix1')}.$$

   Dado que ahora no consideramos `p` en el estrato 1, `Fix1'` se puede calcular correctamente como:

   $$\texttt{Fix1'=\{(q(X),true), (q(X),X=a), (q(X),X=b), (r(X),X=c)\}}.$$

2. Forzado del objetivo `r(X)` con el nuevo punto fijo mediante la llamada:

```
force(Delta1,Stratification,Fix1',r(X),C)
```

que devuelve la restricción `C` como `X=c`.

Tras este paso, `(p(X), X=c)` se añade a la interpretación anterior obteniendo:

```
Fix2 = {(p(X),X=c), (q(X),X=a), (q(X),X=b) (r(X),X=c)}
```

como el punto fijo final para `Delta` que buscamos. □

Con este ejemplo terminamos el segundo capítulo del trabajo, que resume las publicaciones [A.1,A.2,A.3], donde presentamos el lenguaje de bases de datos deductivas $HH_{\neg}(C)$, sus aportaciones, su fundamento teórico y su implementación.

Publicaciones asociadas al capítulo 2

[A.1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez.
Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas.
En *Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09)*, páginas 117–128. ACM Press, 2009.
→ **Página** 116

[A.2] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Incorporating Integrity Constraints to a Deductive Database System.
En *XI Jornadas sobre Programación y Lenguajes, PROLE2011 (SISTEDES)*
editores: Purificación Arenas, Victor M. Gulías y Pablo Nogueira, páginas 141–152, Septiembre, 2011.
→ **Página** 128

[A.3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
An Extended Constraint Deductive Database: Theory and implementation.
The Journal of Logic and Algebraic Programming, volumen 21, páginas 20–52, 2013.
→ **Página** 140

Capítulo 3

Recursión extendida y razonamiento hipotético en sistemas de bases de datos relacionales

En este capítulo presentamos la segunda parte del trabajo: los lenguajes de bases de datos relacionales R-SQL y HR-SQL junto con sus fundamentos semánticos y sus implementaciones. Los fundamentos teóricos proporcionan semántica a estos dos lenguajes, ambos basados en el lenguaje de consulta estructurado SQL (Structured Query Language por sus siglas en inglés). La novedad que aporta R-SQL es la incorporación de una extensión a SQL estándar para admitir relaciones definidas usando recursión no lineal y recursión mutua. Por su parte, HR-SQL además de la recursión extendida incluye definición de vistas y consultas hipotéticas. Hemos implementado en SWI-Prolog dos sistemas para los lenguajes R-SQL y HR-SQL respectivamente como capas adicionales sobre sistemas gestores de bases de datos relacionales existentes. Estos sistemas procesan las bases de datos de estos lenguajes de acuerdo con su semántica propuesta y materializan sus relaciones en tablas de bases de datos de los sistemas gestores.

3.1. Introducción

El lenguaje de consulta estructurado SQL (Structured Query Language) es el lenguaje estándar de definición y consulta de bases de datos relacionales. SQL se presentó como un lenguaje declarativo que carecía de recursión en sus orígenes [51]. El primer trabajo que encontramos como fundamento del modelo relacional es el AR que Codd presentó en los años 70 [28]. Actualmente los sistemas de bases de datos que utilizan el lenguaje SQL (que se fundamenta en el AR) se ajustan al estándar ANSI/ISO [36] y soportan recursión de forma parcial dado que no permiten recursión no lineal ni recursión mutua.

Otra de las limitaciones expresivas del estándar es que no es posible definir relaciones hipotéticas como lo hacen algunos de los sistemas de bases de datos deductivas [101, 14]. En el campo relacional podemos encontrar trabajos sobre consultas hipotéticas en entornos de procesamiento analítico de datos online (OLAP) [7, 133], *business intelligence* [38], y comer-

cio electrónico [132]. Estos trabajos permiten asumir un conjunto de tuplas como hipótesis en el contexto de una consulta.

En este capítulo presentamos dos extensiones del lenguaje SQL para superar algunas limitaciones expresivas del estándar. En primer lugar *R-SQL* extiende a SQL con un tratamiento más general de la recursión permitiendo definiciones recursivas no lineales y recursión mutua. En segundo lugar *HR-SQL* extiende a *R-SQL* permitiendo manejar información hipotética tanto en vistas como en consultas. Esto supone una novedad dado que se añaden las hipótesis a un lenguaje basado en SQL. Ambos lenguajes están inspirados en el lenguaje $HH\neg(\mathcal{C})$ que presentamos en el capítulo anterior. Además hemos desarrollado dos implementaciones en SWI-Prolog [129] para los dos lenguajes que proponemos.

Publicaciones

A continuación presentamos las publicaciones que fundamentan este capítulo:

- En [B.1] proponemos el lenguaje *R-SQL* cuyo objetivo es superar las limitaciones de la recursión del estándar. La idea tras esta publicación es adaptar técnicas propias de las bases de datos deductivas, como por ejemplo la semántica de punto fijo estratificada, para definir la semántica de una base de datos relacional extendida. Nos adherimos además al modelo original del AR [28] que evita duplicados y valores *null* (véase también [30]). En esta memoria se presenta un sistema práctico para *R-SQL* basado en su semántica de punto fijo. Dicho sistema se articula como una capa adicional sobre un sistema gestor de bases de datos relacional (en adelante SGBDR). Esta capa está implementada en SWI-Prolog, el SGBDR es PostgreSQL y usamos Python como lenguaje de comunicación entre Prolog y la base de datos de PostgreSQL. El sistema genera programas Python (a los que nos referimos como *scripts* según la nomenclatura inglesa) que consisten en instrucciones imperativas junto con instrucciones SQL. La ejecución de los programas Python automatizan el acceso al SGBDR y mediante bucles se encargan de calcular el punto fijo de la base de datos siguiendo la semántica propuesta. El resultado de la ejecución de estos *scripts* es la materialización de las tuplas correspondientes a las relaciones *R-SQL* en tablas del SGBDR.

- En [B.3] extendemos y mejoramos el sistema *R-SQL*. Se trata de un artículo centrado íntegramente en el sistema, su implementación y el análisis de su rendimiento. En él definimos una nueva estratificación maximizando el número de estratos para mejorar la eficiencia (véase la sección 3.4.2). Además de otras mejoras menores, simplificamos el cómputo del punto fijo de la base de datos extrayendo de los bucles el caso base de las definiciones recursivas.

- Finalmente en [B.2] abordamos nuestra propuesta para las consultas hipotéticas sobre bases de datos relacionales [50] presentando el esquema *HR-SQL*. Al igual que *R-SQL*, la implementación del sistema *HR-SQL* está fundamentada por una semántica de punto fijo estratificada. Esta implementación se articula también como una capa sobre el SGBDR DB2 de IBM, en su versión de libre distribución. Implementamos el cómputo de punto fijo de las relaciones utilizando un programa del lenguaje de cuarta generación integrado en DB2: SQL PL.

Continuamos presentando las contribuciones de estas publicaciones.

Contribuciones

La primera aportación de esta parte de la tesis consiste en la formalización y diseño de dos sistemas de bases de datos relacionales, *R-SQL* y *HR-SQL*, que extienden los SGBDR actuales con recursión generalizada, en particular recursión mutua y recursión no lineal (en el lenguaje *R-SQL*) incluyendo además el razonamiento hipotético en vistas y consultas (en el lenguaje *HR-SQL*). La incorporación del razonamiento hipotético se hace a partir de R-SQL, extendiendo la sintaxis con la sentencia `assume` para incluir hipótesis en las consultas, lo que permite además definir vistas que combinen recursión e hipótesis.

En cuanto a la formalización, se aporta una semántica formal para el modelo relacional incorporando técnicas utilizadas habitualmente para fundamentar las bases de datos deductivas [122]. Se ha definido una semántica de punto fijo por estratos para ambos lenguajes.

A partir del marco teórico desarrollado hemos diseñado sendas implementaciones en SWI-Prolog para los lenguajes *R-SQL* y *HR-SQL*, que se incorporan como capas adicionales sobre los SGBDR existentes, extendiéndolos así con nuevas funcionalidades. En concreto, el primero extiende PostgreSQL y el segundo DB2. Además hemos implementado también un prototipo de *R-SQL* sobre MySQL dado que este SGBDR no permite recursión en sus consultas.

Los sistemas *R-SQL* y *HR-SQL* generan *scripts* en Python y SQL PL respectivamente para generar nuevas tablas con las tuplas correspondientes a su semántica propuesta en el SGBDR. Además *HR-SQL* utiliza tablas temporales para computar el resultado de vistas y consultas con hipótesis. Este tipo de tablas tienen la ventaja de ser más eficientes desde el punto de vista del rendimiento y además tras el cómputo son descartadas por lo que su uso resulta adecuado para este fin. *HR-SQL* utiliza la noción de dependencias entre relaciones que proviene de las BDD [122] para determinar qué tablas deben ser recalculadas.

Finalmente, proponemos también mejoras en el cálculo del punto fijo de las relaciones *HR-SQL* con respecto a *R-SQL* que optimimizan el rendimiento del primer sistema (véase la sección 3.6).

3.2. Extendiendo SQL

Comenzamos esta sección presentando el lenguaje de definición de bases de datos de *R-SQL* y de *HR-SQL*. Para la definición de las relaciones de la base de datos ambos lenguajes utilizan la misma sintaxis, por ello a lo largo de esta sección nos referiremos al lenguaje *HR-SQL*, entendiendo que en todo lo referente a definición de relaciones es válido también para *R-SQL*.

En las subsecciones 3.2.1 y 3.2.2 presentamos respectivamente el lenguaje de consulta y el lenguaje de definición de vistas que son propios únicamente de *HR-SQL* que a su vez extiende a *R-SQL* como se presenta en [B.2].

El lenguaje *HR-SQL* utiliza una sintaxis muy similar a SQL. Sin embargo, a diferencia del estándar hemos realizado algunos cambios para introducir relaciones recursivas cuando se definen las bases de datos. La definición de una base de datos consiste en asignaciones de instrucciones de consulta de SQL estándar (a la que nos referimos como instrucción *select* o `sel_stm`) a nombres de relaciones (`R`) junto a los nombres y tipos de sus campos (a lo que nos referiremos como esquema de la relación o `sch`). Así, una definición de relación en *HR-SQL* es de la forma:

$$\texttt{R sch} := \texttt{sel\_stm};$$

donde `R` puede aparecer en `sel_stm`, y en este caso se trataría de una definición recursiva.

Una base de datos **db** es una secuencia no vacía de definiciones de relaciones que pueden ser por tanto recursivas. La sintaxis formal para las bases de datos *HR-SQL* se define usando las reglas gramaticales que aparecen en la figura 3.1.

```
db       ::=  R sch := sel_stm; ... R sch := sel_stm;
sch      ::=  (A T,...,A T)
sel_stm  ::=  select exp,...,exp [from R,...,R [where cond]]
              sel_stm union sel_stm  |  sel_stm except sel_stm
exp      ::=  C | R.A | exp m_op exp | -exp
cond     ::=  true | false | exp b_op exp | not cond |
              cond [and|or] cond
m_op     ::=  + | - | / | *
b_op     ::=  = | <>| <| >| >= | <=
```

Figura 3.1: Reglas gramaticales del lenguaje *HR-SQL*

En la figura, además de las categorías gramaticales ya introducidas, **T** representa los tipos SQL que podemos encontrar en el estándar como **integer**, **float**, **varchar(n)**; **cond** representa condiciones booleanas; **m_op** y **b_op** representan respectivamente operadores matemáticos y booleanos; y **C** representa cualquier constante válida de SQL.

Además utilizamos los corchetes para representar que el fragmento encerrado entre ellos es opcional y **R.A** representa un atributo **A** calificado con la relación **R** a la que pertenece. Al igual que SQL, utilizamos * como *azúcar sintáctico* para representar la lista de proyección de todos los atributos de las relaciones de la cláusula **from** de una instrucción **sel_stm**. Por legibilidad, utilizamos minúsculas para representar las palabras reservadas de *HR-SQL* que provienen de SQL (como **select**, **where**, **from**) y aparecen en mayúscula en los ejemplos de SQL estándar.

A continuación introducimos unos conjuntos de relaciones que nos serán útiles al definir la semántica del lenguaje.

- $\mathrm{RN}_{\mathtt{db}}$ representa el conjunto de relaciones $\{\mathtt{R}_1, \ldots, \mathtt{R}_n\}$ definidas en la base de datos **db**.
- $\mathrm{RN}_{\mathtt{sel\_stm}}$ representa el conjunto de nombres de relación que aparece en **sel_stm**.
- Para el caso $\mathtt{sel\_stm} = \mathtt{sel\_stm}_1$ **except** $\mathtt{sel\_stm}_2$ también definimos $\mathrm{RN}^{\neg}_{\mathtt{sel\_stm}}$ como el conjunto de relaciones que aparece en $\mathtt{sel\_stm}_2$ (nótese que $\mathrm{RN}^{\neg}_{\mathtt{sel\_stm}} \subseteq \mathrm{RN}_{\mathtt{sel\_stm}}$).

Suponemos que para todo **R sch:= sel_stm** definido en **db** se verifica que $\mathrm{RN}_{\mathtt{sel\_stm}} \subseteq \mathrm{RN}_{\mathtt{db}}$.

Proponemos a continuación un ejemplo con otra aplicación del cierre transitivo de un grafo a una base de datos de viajes que combina diferentes medios de transporte. Extenderemos este ejemplo a lo largo del capítulo para mostrar distintas ventajas de nuestro lenguaje.

Ejemplo 14 La base de datos que mostramos a continuación está inspirada en un ejemplo que aparece en [24] y representa los posibles medios de transporte que aparecen en la figura 3.2 para la gestión de viajes en las Islas Canarias. Estos viajes pueden ser entre distintas islas o dentro de ellas.

En concreto, en esta base de datos se incluyen relaciones para vuelos (**flight**), autobuses (**bus**) y barcos (**boat**) todas con el siguiente esquema:

(**ori** varchar(10), **des** varchar(10), **time** float)

que almacena información sobre el origen (**ori**), destino (**des**) y duración (**time**) de una serie de trayectos posibles en las Islas Canarias.

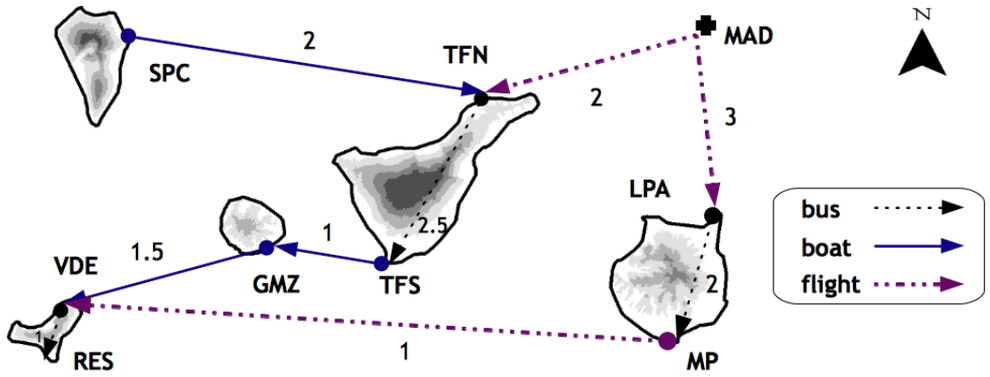


Figura 3.2: Representación gráfica de los medios de transporte posibles en las Islas Canarias para el ejemplo 14.

La relación **link** incluye los posibles itinerarios utilizando cualquiera de los medios de transporte anteriores:

```
link(ori varchar(10), des varchar(10), time float):=
  select * from flight union
  select * from boat union
  select * from bus;
```

La relación **travel** es el cierre transitivo de **link**, i.e., nos dará información acerca de posibles viajes utilizando los transportes de la base de datos, que pueden (o no) concatenar varios medios de transporte. Mostramos a continuación su definición en la sintaxis propuesta.

```
travel(ori varchar(10), des varchar(10), time float):=
  select * from link union
  select link.ori, travel.des, link.time + travel.time
    from link,travel
    where link.des = travel.ori;
```

Con esta relación obtenemos con la aparición de **travel** en su propia definición una tabla con los posibles itinerarios de viaje y su duración correspondiente. □

Por su parte, SQL-99 [36] admite definiciones recursivas de vistas mediante la cláusula WITH RECURSIVE como presentamos a continuación mediante un ejemplo. La parte extensional de la siguiente base de datos incluye las tablas **mother** y **father** con esquema (**parent varchar(20), child varchar(20)**) que representan respectivamente que el primer atributo es la madre o el padre del segundo. Para establecer la relación de antepasado en SQL debemos definir en primer lugar la vista auxiliar **parent**:

```
CREATE VIEW parent(parent,child) AS
  SELECT * FROM mother
   UNION
  SELECT * FROM father;
```

La vista **rec_ancestor** establece la relación de antepasado:

```
CREATE OR REPLACE VIEW ancestor(ancestor,descendant) AS
  WITH RECURSIVE rec_ancestor(ancestor,descendant) AS
    SELECT * FROM parent
      UNION ALL
    SELECT parent,descendant
      FROM parent, rec_ancestor
      WHERE parent.child=rec_ancestor.ancestor
  SELECT * FROM rec_ancestor;
```

Se trata de una vista recursiva dado que se referencia a si misma en la cláusula **FROM** de su definición. Las implementaciones de SQL no admiten más de un caso base en la definición recursiva. Esta limitación no se da en nuestro lenguaje y podemos formular la misma vista **rec_ancestor** como:

```
rec_ancestor(ancestor varchar(10), descendant varchar(10)):=
  select * from mother union
  select * from father union
  select ancestor, descendant
    from mother, father, rec_ancestor
    where father.descendant=rec_ancestor.ancestor or
          mother.descendant=rec_ancestor.ancestor;
```

Como vemos, la formulación SQL es más compleja porque, entre otras cosas, exige añadir una relación auxiliar cuando hay varios casos base (**parent** en el ejemplo). Además las definiciones recursivas solo pueden aparecer en vistas y no en las relaciones de la base de datos. La definición de **rec_ancestor** en *HR-SQL* es más sencilla, legible y compacta.

Podemos formular la vista **travel** del ejemplo 14 utilizando sintaxis SQL estándar:

```
CREATE OR REPLACE VIEW travel(ori,des,time) AS
  WITH RECURSIVE rec_travel(ori,des,time) AS
    SELECT link.* FROM link
      UNION
    SELECT rec_travel.ori,link.des, link.time + rec.travel.time
      FROM rec_travel,link
      WHERE rec_travel.des= link.ori
  SELECT * FROM rec_travel;
```

Sin embargo, esta vista no tiene sentido en SQL dado que el estándar exige **UNION ALL** para preservar duplicados cuando se utiliza la palabra reservada **WITH RECURSIVE** [102]. Por tanto, esta vista sería rechazada por DB2, Oracle y PostgreSQL. Por otro lado, ni Access ni MySQL proporcionan recursión en SQL.

Otra de las limitaciones de SQL-99 es que no permite recursión mutua. En el siguiente ejemplo mostramos cómo se puede escribir en *HR-SQL* un sencillo ejemplo de dos relaciones mutuamente recursivas.

Ejemplo 15 Las siguientes relaciones **even** y **odd** representan respectivamente la secuencias de números pares e impares hasta 100.

```
even(x integer) :=
  select 0 union
  select odd.x+1 from odd where odd.x<100;

odd(x integer) :=
  select even.x+1 from even where even.x<100;
```

Notése que **select 0** es una instrucción SQL *from-less* (aceptada en muchos SGBDR) y que sencillamente devuelve la tupla especificada por las expresiones de la lista de proyección (0 en este caso). En la memoria utilizamos frecuentemente este tipo de instrucciones *from-less* para especificar los casos base de las definiciones recursivas. Otra alternativa que proporcionan los SGBDR (como Oracle) para incluir este tipo de instrucciones es hacer referencia a la tabla *dual* del sistema gestor en la cláusula **FROM** de la instrucción *select* (**select 0 from dual**). *HR-SQL* admite ambas formulaciones para definir sus casos base. Encontramos más ejemplos de relaciones con recursión no lineal y recursión mutua en la sección 2 de [B.1]. □

3.2.1. El lenguaje de consulta

Las consultas del lenguaje *R-SQL*, dado que no incluyen hipótesis, se limitan a la categoría **sel_stm** que aparece en la figura 3.2. Para formular consultas hipotéticas en *HR-SQL* extendemos las instrucciones *select* de *R-SQL*. La semántica pretendida para una consulta hipotética es el resultado de la consulta a la base de datos cuyas definiciones han sido extendidas con las suposiciones que aparecen tras la palabra reservada **assume**.

A continuación en la figura 3.3 extendemos la gramática presentada de la figura 3.2 para incorporar el lenguaje de consulta que permite hipótesis.

query	::=	sel_stm | sel_hyp
sel_hyp	::=	assume hypo, ... , hypo sel_stm
hypo	::=	sel_stm [not] in R

Figura 3.3: El lenguaje de consulta de *HR-SQL*

Ejemplo 16 Un ejemplo de consulta *HR-SQL* para la base de datos del ejemplo 14 es: ¿cuánto se tardaría desde Madrid hasta Valverde, suponiendo que no viajamos en ninguno de los itinerarios en barco que tardan más de una hora? Esta consulta se expresa en *HR-SQL* como:

```
assume select * from boat where boat.time>1 not in link
(select travel.time from travel
   where travel.ori = 'MAD' and travel.des = 'VDE');
```

En esta consulta **select * from boat where boat.time>1 not in link** constituye la hipótesis (**hypo**) que afecta a la instrucción *select* (**sel_stm**) y que escribimos entre paréntesis por legibilidad. □

3.2.2. El lenguaje de definición de vistas

En esta sección explicamos cómo *HR-SQL* extiende el lenguaje de definición para permitir la definición de vistas que pueden (o no) ser hipotéticas. Utilizamos el lenguaje de consulta de la subsección anterior y definimos las vistas de forma similar a las relaciones, dándoles un nombre (esta vez sin esquema) y asignando a estos nombres una `query` para permitir referencias desde otra vista o consulta e incluso recursión.

La aproximación que seguimos para el lenguaje de definición de vistas hipotéticas es similar a la que sigue SQL para la recursión permitiendo la definición de vistas recursivas mediante `WITH RECURSIVE`. Permitimos el razonamiento hipotético mediante `assume` una vez que la base de datos ya está calculada y distinguimos entre relaciones que definen la base de datos y vistas en las que se pueden formular las suposiciones.

En adelante usamos $\mathtt{V}$ para representar nombres de vistas que se definen mediante una consulta no hipotética a las que se les asigna un `sel_stm` (y se definen de la misma forma que las relaciones en la base de datos). Usamos $\mathtt{HV}$ para las vistas hipotéticas a las que se les asigna un `sel_hyp`.

En la figura 3.4 se muestra la sintaxis para la definición de vistas.

```
vd    ::=  view ... view
view  ::=  V sch := sel_stm; |
           HV sch := sel_hyp;
```

Figura 3.4: El lenguaje de definición de vistas de *HR-SQL*

Permitimos la recursión mutua solo para secuencias de definiciones de vistas no hipotéticas para evitar que las llamadas de la recursión mutua junto con las hipótesis conlleve un cómputo no terminante. No obstante, una vez declarada una secuencia de vistas mutuamente recursivas, se puede invocar el nombre de una de estas vistas dentro de una nueva vista hipotética.

Una *secuencia de definiciones de vistas* para la base de datos `db`, denotada como `vd`, es una secuencia de la forma que aparece a continuación, tal que los nombres de relación que aparecen ella son o bien nombres de relación de `db` o bien nombres de vistas de `vd`.

$$
\begin{array}{lll}
\mathtt{V}_1\ \mathtt{sch}_1 & ::= & \mathtt{sel\_stm}_1; \\
 & \ldots & \\
\mathtt{V}_m\ \mathtt{sch}_m & ::= & \mathtt{sel\_stm}_m; \\
\mathtt{HV}_1\ \mathtt{sch}_1 & ::= & \mathtt{sel\_hyp}_1; \\
 & \ldots & \\
\mathtt{HV}_r\ \mathtt{sch}_r & ::= & \mathtt{sel\_hyp}_r;
\end{array}
$$

Nótese que definimos en primer lugar las vistas normales y en segundo lugar las hipotéticas. Además para evitar secuencias de definiciones de vistas mutuamente recursivas e hipotéticas al mismo tiempo, imponemos las siguientes condiciones:

- Para todo $j = 1..m$, $\mathtt{V}_j$ puede aparecer en cualquier vista. Sin embargo, $\mathtt{V}_j$ no puede aparecer en una instrucción `assume`.
- Para todo $j = 1..r$, $\mathtt{HV}_j$ puede aparecer dentro de la instrucción *select* de su propia definición $sel(\mathtt{sel\_hyp}_j)$, pero no en $\mathtt{sel\_stm}_1, \ldots, \mathtt{sel\_stm}_m$, $\mathtt{sel\_hyp}_k$, si $k \neq j$, ni en la parte asumida de $\mathtt{sel\_hyp}_j$.

A continuación mostramos un ejemplo de definición de una vista hipotética usando el lenguaje de definición de vistas de *HR-SQL*.

Ejemplo 17 A partir del ejemplo 14 planteamos una nueva vista hipotética:

Suponemos que hay una erupción volcánica en El Hierro que conlleva el cierre del espacio aéreo y la eliminación del autobús en la isla. Además se añade un barco desde El Hierro hasta La Palma ¿Qué lugares del archipiélago son alcanzables bajo estas suposiciones?

Para entender mejor esta consulta podemos ver los trayectos en la figura 3.5. En ella coloreamos los trayectos añadidos debido las suposiciones en color rojo. Además, distinguimos las suposiciones negativas (`not in`) de las suposiciones positivas (`in`) utilizando trazo discontinuo para las negativas y continuo para las positivas.

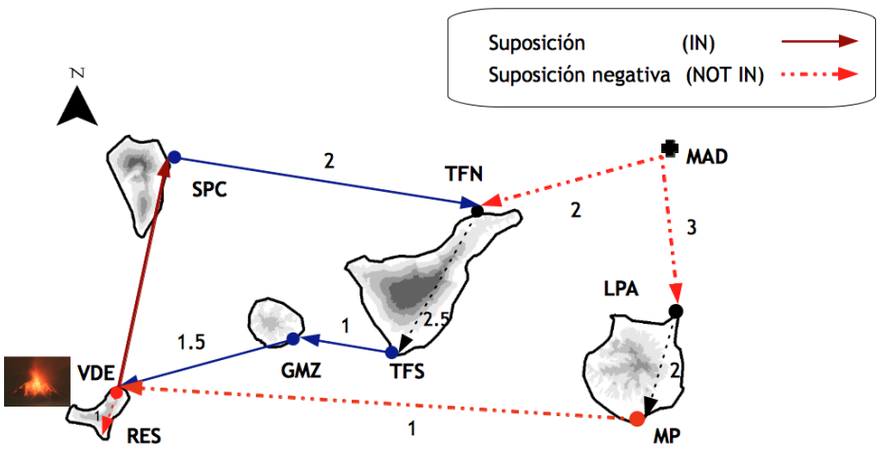


Figura 3.5: Representación gráfica de la vista del ejemplo 17.

Esta consulta se puede expresar usando el lenguaje de definición de vistas de *HR-SQL* como:

```
reachable(ori varchar(10),des varchar(10)) :=
 assume (select * from bus where bus.ori = 'VDE' union
         select * from flight not in link),
         select 'RES','SPC',1.5 in boat
 select link.ori, link.des from link union
 select link.ori, reachable.des from link, reachable
         where link.des = reachable.ori
```

En esta vista combinamos las hipótesis con el cierre transitivo de los itinerarios. A continuación mostramos las tuplas resultado:

```
[('TNF', 'LC'),  ('GC', 'MP'),   ('LC', 'GOM'),  ('LP', 'TNF'),
 ('VAL', 'RES'), ('GOM', 'VAL'), ('TNF', 'GOM'), ('LP', 'LC'),
 ('GOM', 'RES'), ('LC', 'VAL'),  ('LP', 'GOM'),  ('TNF', 'VAL'),
 ('LC', 'RES'),  ('TNF', 'RES'), ('LP', 'VAL'),  ('LP', 'RES')]
```

Como hemos señalado, se corresponden con todos los trayectos posibles entre las ciudades de las islas en caso de la erupción volcánica. □

3.3. Fundamentos teóricos

En esta sección se tratan los fundamentos teóricos del lenguaje SQL extendido que se pueden encontrar en las publicaciones [B.1,B.2]. En [B.1] proponemos una semántica de punto fijo estratificada para el lenguaje *R-SQL* (sin razonamiento hipotético). En [B.2] proponemos además un lenguaje propio de definición para vistas y un lenguaje de consulta. Para el cálculo de la semántica de los lenguajes de definición, de vistas y de consulta usamos técnicas de punto fijo estratificado al igual que hacíamos con el lenguaje $HH\neg(C)$. Como es sabido la estratificación se basa en la construcción de un grafo de dependencias [122]. La noción de grafo de dependencias entre relaciones ha resultado útil a la hora de incorporar el razonamiento hipotético al lenguaje *HR-SQL* dado que se utiliza para identificar qué relaciones deben ser recalculadas cuando se define una vista o se plantea una consulta.

A continuación presentamos los fundamentos semánticos de *HR-SQL* y *R-SQL*. Como es habitual para lenguajes de bases de datos relacionales, el significado de cada relación en una base de datos de se corresponde con el conjunto de tuplas que satisfacen su definición. En las siguientes subsecciones defininiremos la semánticas para vistas y consultas hipotéticas propia solamente de *HR-SQL*.

3.3.1. Semántica para las bases de datos

Comenzaremos la sección de fundamentos teóricos presentando la definición del grafo de dependencias para una base de datos. En adelante denotamos como $\mathtt{RN}_{\mathtt{db}}$ al conjunto de nombres de relación definidos en la base de datos **db**.

Definición 8 El *grafo de dependencias* asociado a una base de datos **db**, que denotamos como $DG_{\mathtt{db}}$, es un grafo dirigido donde:

- Los nodos de $DG_{\mathtt{db}}$ son los elementos del conjunto $\mathtt{RN}_{\mathtt{db}}$ y
- los arcos de $DG_{\mathtt{db}}$ los definimos como sigue:
  - Para toda relación $\mathtt{R}$ `sch := sel_stm` habrá un arco desde cada nombre de relación $\mathtt{R}' \in \mathtt{RN}_{\mathtt{sel\_stm}}$ hacia $\mathtt{R}$.
  - Estarán *etiquetados negativamente* todos los arcos que parten de nombres de relaciones que pertenecen a $\mathtt{RN}^{\neg}_{\mathtt{sel\_stm}}$.

Para todo par de relaciones $\mathtt{R}_1, \mathtt{R}_2 \in \mathtt{RN}_{\mathtt{db}}$, diremos que $\mathtt{R}_2$ *depende* de $\mathtt{R}_1$ si hay un camino desde $\mathtt{R}_1$ hasta $\mathtt{R}_2$ en $DG_{\mathtt{db}}$. Y $\mathtt{R}_2$ *depende negativamente* de $\mathtt{R}_1$ si hay un camino desde $\mathtt{R}_1$ hasta $\mathtt{R}_2$ en $DG_{\mathtt{db}}$ con, al menos, un arco negativo. □

Estratificación

A continuación, usando la definición de $DG_{\mathtt{db}}$ de la subsección anterior, definimos el concepto de estratificación para una base de datos **db**.

Definición 9 Una *estratificación* de una base de datos **db** compuesta por n relaciones, es una función $str : \mathtt{RN}_{\mathtt{db}} \rightarrow \{1, \ldots, n\}$ tal que:

- $str(\mathtt{R}_i) \leq str(\mathtt{R}_j)$, si $\mathtt{R}_j$ depende de $\mathtt{R}_i$ y
- $str(\mathtt{R}_i) < str(\mathtt{R}_j)$, si $\mathtt{R}_j$ depende negativamente de $\mathtt{R}_i$.

Diremos que una base de datos $\mathtt{db}$ es estratificable si existe una estratificación para ella. Además, llamaremos a $str(\mathtt{R})$ el *estrato* de $\mathtt{R}$. Para las instrucciones $\mathtt{sel\_stm}$ definimos su estrato como $str(\mathtt{sel\_stm}) = max\{str(\mathtt{R}_i) \mid \mathtt{R}_i \in \mathtt{RN}_{\mathtt{sel\_stm}}\}$. □

Para las nociones teóricas utilizamos una base de datos concreta que denotamos como $\mathtt{db}$ y una estratificación para ella, denotada como str. Suponemos también que cada relación $\mathtt{R}$ tiene su tupla de atributos $\mathtt{A}_i$ y tipos $\mathtt{T}_i$ asociados (que se corresponde con el esquema de la relación $\mathtt{R}$), que escribimos $\mathtt{R}$ $(\mathtt{A}_1\ \mathtt{T}_1, ..., \mathtt{A}_r\ \mathtt{T}_r)$, de forma que cada tipo $\mathtt{T}_i$, $i = 1..r$ representa su dominio correspondiente D_i. Por ejemplo el tipo $\mathtt{integer}$ representa el dominio de los números enteros.

Denotamos también con $\mathcal{D}$ el *dominio universal* que es la unión de todos los dominios de los tipos de las base de datos. Dado que las relaciones pueden tener diferentes aridades, usaremos el conjunto $\mathcal{T} = \bigcup_{n\geq 1} \mathcal{D}^n$, en donde tomarán valores las tuplas de las relaciones de $\mathtt{db}$.

De nuevo, como hacemos en la sección 2.2.2 del capítulo 2, usamos el concepto de interpretación para dar semántica a las bases de datos de *HR-SQL*. Las interpretaciones son funciones que asocian un elemento de $\mathcal{P}(\mathcal{T})$ (partes de $\mathcal{T}$) a cada una de las relaciones de $\mathtt{RN}_{\mathtt{db}}$ y también se clasifican por estratos. La siguiente definición formaliza la noción de interpretación:

Definición 10 Sea $i \geq 1$, una *interpretación* I para una base de datos $\mathtt{db}$ sobre un estrato i es una función $I : \mathtt{RN}_{\mathtt{db}} \to \mathcal{P}(\mathcal{T})$, tal que para todas las relaciones $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$ con esquema $\mathtt{sch}$ se cumple que:

- Si $\mathtt{sch} \equiv (\mathtt{A}_1\ \mathtt{T}_1, \ldots, \mathtt{A}_r\ \mathtt{T}_r)$, y $D_1, \ldots, D_r$ son respectivamente los dominios asociados a los tipos $\mathtt{T}_1, \ldots, \mathtt{T}_r$ entonces $I(\mathtt{R}) \subseteq D_1 \times \ldots \times D_r$,
- $I(\mathtt{R}) = \emptyset$, en caso de que $str(\mathtt{R}) > i$.

La semántica de las bases de datos *HR-SQL* se calcula estrato por estrato. Cada interpretación se corresponde con el significado de un estrato. Al conjunto de interpretaciones de $\mathtt{db}$ sobre el estrato i le llamamos $\mathcal{I}_i^{\mathtt{db}}$. Pasamos a definir la relación de orden entre interpretaciones:

Sea $I_1, I_2 \in \mathcal{I}_i^{\mathtt{db}}$. *$I_1$ es menor o igual que I_2 en el estrato i*, (denotado $I_1 \sqsubseteq_i I_2$), si se satisfacen las siguientes condiciones para todo $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$:

- $I_1(\mathtt{R}) = I_2(\mathtt{R})$, si $str(\mathtt{R}) < i$, y
- $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$, si $str(\mathtt{R}) = i$. □

Para todo i, $(\mathcal{I}_i^{\mathtt{db}}, \sqsubseteq_i)$ es un conjunto parcialmente ordenado. La idea de este conjunto es que cuando una interpretación sobre un estrato cualquiera i crece, su conjunto de tuplas asociado puede incrementarse también. Sin embargo, los conjuntos de tuplas asociados a relaciones de estratos inferiores permanecen invariables. Además $(\mathcal{I}_i^{\mathtt{db}}, \sqsubseteq_i)$ es un conjunto completo parcialmente ordenado: si $\{I_n\}_{n\geq 0}$ es una cadena en $(\mathcal{I}_i^{\mathtt{db}}, \sqsubseteq_i)$, entonces $\hat{I}$, definido como $\hat{I}(\mathtt{R}) = \bigcup_{n\geq 0} I_n(\mathtt{R})$, $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$, es la menor de las cotas superiores de $\{I_n\}_{n\geq 0}$.

La siguiente definición formaliza el significado de $\mathtt{sel\_stm}$ en el contexto de una interpretación I.

Definición 11 Sea $i \geq 1$, $I \in \mathcal{I}_i^{\mathtt{db}}$. Sea $\mathtt{sel\_stm}$ una instrucción *select*, tal que $str(\mathtt{sel\_stm}) \leq i$. Definimos recursivamente la *interpretación de* $\mathtt{sel\_stm}$ *con respecto a* I para $\mathtt{db}$, denotada con $[\![\mathtt{sel\_stm}]\!]^I$, de la siguiente forma:

- $[\![\texttt{select exp}_1, \ldots, \texttt{exp}_k]\!]^I = \{(exp_1, \ldots, exp_k)\}$,
  donde exp_i representa la evaluación de $\texttt{exp}_i$.
- $[\![\texttt{select exp}_1, \ldots, \texttt{exp}_k \texttt{ from } \mathtt{R}_1, \ldots, \mathtt{R}_m \texttt{ where cond}]\!]^I =$
  $\{(exp_1[\overline{a}/\overline{\mathtt{A}}], \ldots, exp_k[\overline{a}/\overline{\mathtt{A}}]) \mid \overline{a} \in I(\mathtt{R}_1) \times \ldots \times I(\mathtt{R}_m)$ y se satisface $cond[\overline{a}/\overline{\mathtt{A}}]\}$,

  donde $\overline{\mathtt{A}}$ representa una secuencia de atributos (prefijados con la relación a la que pertenecen). Si $\mathtt{A}^j_1, \ldots, \mathtt{A}^j_{r_j}$ son los atributos de $\mathtt{R}_j$, $1 \leq j \leq m$, entonces:

  - $\overline{\mathtt{A}}$ es la secuencia completa de $\mathtt{R}_1.\mathtt{A}^1_1, \ldots, \mathtt{R}_1.\mathtt{A}^1_{\mathtt{r}_1}, \ldots, \mathtt{R}_m.\mathtt{A}^{\mathtt{m}}_1, \ldots, \mathtt{R}_m.\mathtt{A}^{\mathtt{m}}_{\mathtt{r}_\mathtt{m}}$;
  - la notación $exp_j[\overline{a}/\overline{\mathtt{A}}]$, $1 \leq j \leq k$, representa la evaluación de $\texttt{exp}_j$, una vez reemplazado $\overline{A}$ por $\overline{\mathtt{a}}$ en $\texttt{exp}_j$; y
  - $cond[\overline{a}/\overline{\mathtt{A}}]$ representa la evaluación lógica de **cond** con la sustitución anterior.
- $[\![\texttt{sel\_stm}_1 \texttt{ union sel\_stm}_2]\!]^I = [\![\texttt{sel\_stm}_1]\!]^I \cup [\![\texttt{sel\_stm}_2]\!]^I$, donde $\cup$ representa la unión de conjuntos.
- $[\![\texttt{sel\_stm}_1 \texttt{ except sel\_stm}_2]\!]^I = [\![\texttt{sel\_stm}_1]\!]^I \setminus [\![\texttt{sel\_stm}_2]\!]^I$, donde $\setminus$ representa la diferencia de conjuntos. □

Para todo i definimos un operador $T^{\mathtt{db}}_i$ sobre el conjunto $\mathcal{I}^{\mathtt{db}}_i$ de interpretaciones del estrato i para **db**. Este operador es continuo como vemos en la proposición 2. De forma análoga a como procedemos en $HH\neg(\mathcal{C})$, el mínimo punto fijo de $T^{\mathtt{db}}_i$ es la interpretación que da significado a todas las relaciones de la base de datos **db** en el estrato i. A diferencia de lo que ocurría en la semántica de punto fijo del capítulo anterior, calificaremos el operador con la base de datos **db** dado que el operador se aplica a una base de datos concreta. Usando de nuevo el teorema de Knaster-Tarski (como hicimos en la sección 2.2.2), el punto fijo se puede obtener como el supremo de la cadena de interpretaciones que obtenemos mediante la aplicación sucesiva de este operador, partiendo de una interpretación mínima. El operador $T^{\mathtt{db}}_i$ se define a continuación.

Definición 12 El *operador* $T^{\mathtt{db}}_i : \mathcal{I}^{\mathtt{db}}_i \longrightarrow \mathcal{I}^{\mathtt{db}}_i$ transforma interpretaciones sobre i de la siguiente forma. Para todo $I \in \mathcal{I}^{\mathtt{db}}_i$ y para todo $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$:

- $T^{\mathtt{db}}_i(I)(\mathtt{R}) = I(\mathtt{R})$, si $str(\mathtt{R}) < i$.
- $T^{\mathtt{db}}_i(I)(\mathtt{R}) = [\![\texttt{sel\_stm}]\!]^I$, si $str(\mathtt{R}) = i$ y **sel_stm** es la definición de $\mathtt{R}$ en **db**.
- $T^{\mathtt{db}}_i(I)(\mathtt{R}) = \emptyset$, si $str(\mathtt{R}) > i$. □

Este operador $T^{\mathtt{db}}_i$ es continuo como se enuncia a continuación.

Proposición 2 (Continuidad de $T^{\mathtt{db}}_i$) Sea $i \geq 1$ y $\{I_n\}_{n\geq 0}$ una cadena de interpretaciones sobre $\mathcal{I}^{\mathtt{db}}_i$ ($I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$). Entonces, $T^{\mathtt{db}}_i(\bigsqcup_{n\geq 0} I_n) = \bigsqcup_{n\geq 0} T^{\mathtt{db}}_i(I_n)$.

Por tanto, la existencia de un mínimo punto fijo que se consigue, estrato por estrato, es una consecuencia directa del teorema de punto fijo de Knaster-Tarski [118] como se muestra en [B.1].

Teorema 2 Existe una interpretación $fix^{\mathtt{db}} : \mathtt{RN}_{\mathtt{db}} \to \mathcal{P}(\mathcal{T})$, tal que para $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$, si **sel_stm** es la definición de $\mathtt{R}$ en **db**, entonces $fix^{\mathtt{db}}(\mathtt{R}) = [\![\texttt{sel\_stm}]\!]^{fix^{\mathtt{db}}}$.

Así pues, la interpretación $fix^{\mathtt{db}}$ nos da la semántica de **db**. La construcción de su punto fijo estrato por estrato se define aplicando el operador de forma sucesiva. Veremos a continuación los casos $T_1^{\mathtt{db}}$ y $T_2^{\mathtt{db}}$ y generalizaremos para $T_n^{\mathtt{db}}$.

El operador $T_1^{\mathtt{db}}$ tiene un mínimo punto fijo $fix_1^{\mathtt{db}}$, que es $\bigsqcup_{n\geq 0}(T_1^{\mathtt{db}})^n(\emptyset)$, el supremo de la cadena $\{(T_1^{\mathtt{db}})^n(\emptyset)\}_{n\geq 0}$, donde $(T_1^{\mathtt{db}})^n(\emptyset)$ es el resultado de las n sucesivas aplicaciones de $T_1^{\mathtt{db}}$, partiendo de la interpretación vacía. Consideramos la secuencia $\{(T_2^{\mathtt{db}})^n(fix_1^{\mathtt{db}})\}_{n\geq 0}$ de interpretaciones en $(\mathcal{I}_2^{\mathtt{db}}, \sqsubseteq_2)$ mayor que $fix_1^{\mathtt{db}}$. Si usamos la definición de $T_i^{\mathtt{db}}$ y teniendo en cuenta que $fix_1^{\mathtt{db}}(\mathtt{R}) = \emptyset$ para todo $\mathtt{R}$ tal que $str(\mathtt{R}) \geq 2$, podemos probar fácilmente (por inducción sobre $n \geq 0$) que esa secuencia es también una cadena, $fix_1^{\mathtt{db}} \sqsubseteq_2 T_2^{\mathtt{db}}(fix_1^{\mathtt{db}}) \sqsubseteq_2 T_2^{\mathtt{db}}(T_2^{\mathtt{db}}(fix_1^{\mathtt{db}})) \sqsubseteq_2 \ldots, \sqsubseteq_2 (T_2^{\mathtt{db}})^n(fix_1^{\mathtt{db}}), \ldots$ con un supremo

$$fix_2^{\mathtt{db}} = \bigsqcup_{n\geq 0}(T_2^{\mathtt{db}})^n(fix_1),$$

que es el mínimo punto fijo de $T_2^{\mathtt{db}}$ que contiene $fix_1^{\mathtt{db}}$, y que llamaremos $fix_2^{\mathtt{db}}$.

Si definimos k como $max\{str(\mathtt{R}) \mid \mathtt{R} \in \mathtt{RN}_{\mathtt{db}}\}$, podemos encontrar, para todo i, $1 < i \leq k$, una cadena:

$$\{(T_i^{\mathtt{db}})^n(fix_{i-1}^{\mathtt{db}})\}_{n\geq 0},$$

y encontramos el punto fijo tal que:

$$fix_i^{\mathtt{db}} = \bigsqcup_{n\geq 0}(T_i^{\mathtt{db}})^n(fix_{i-1}^{\mathtt{db}}),$$

Denominamos $fix^{\mathtt{db}}$ a $fix_k^{\mathtt{db}}$ dado que contiene la semántica del punto fijo de la base de datos **db**.

Una vez formalizada la semántica de la base de datos pasamos presentar en primer lugar la semántica de las consultas y después la semántica del lenguaje de definición de vistas.

3.3.2. La semántica de las consultas

La respuesta a una consulta en *HR-SQL* para una base de datos estratificable **db** es la interpretación de dicha consulta con respecto al punto fijo de la base de datos **db**. Como las consultas en *HR-SQL* pueden ser hipotéticas debemos tener en cuenta que para obtener la respuesta correcta es necesario modificar algunas de las relaciones de la base de datos en estos casos. Desde el punto de vista lógico una consulta hipotética se puede interpretar como una implicación de la lógica intuicionista clásica [72], es decir, representa el valor de un consecuente supuesto el antecedente.

La idea general tras el cálculo de consultas hipotéticas es que para cada consulta se modifican las relaciones *necesarias* de la base de datos (de las que depende la consulta) para reflejar las suposiciones y calcular su respuesta. Estas relaciones necesarias para una consulta concreta son:

- las que aparecen explícitamente tras la palabra reservada **in** (también **not in**) o
- las que dependen de las anteriores según la definición 8, i.e., según el grafo de dependencias.

Para representar los cambios necesarios en la base de datos actual en el caso de una consulta hipotética usaremos la siguiente notación:

$$\texttt{db[R sch := sel\_stm}'\texttt{/R sch := sel\_stm]}$$

que representa la base de datos **db** una vez reemplazada la definición de **R sch := sel_stm** por **R sch := sel_stm**′. También usaremos *sel*(**query**) para representar la sentencia **sel_stm** de la consulta **query** como se especifica a continuación:

- *sel*(**sel_stm**)= **sel_stm** y
- *sel*(**assume hypo**$_1, \ldots,$ **hypo**$_k$ **sel_stm**) = **sel_stm**.

Para facilitar la lectura introducimos a continuación cómo se maneja una sola hipotésis **hypo** dentro de la consulta **query** concreta mediante reemplazamiento. Para resolver la secuencia **hypo**$_1, \ldots,$ **hypo**$_k$ se deben aplicar secuencialmente estos reemplazamientos como explicamos en el ejemplo 18 más adelante.

La siguiente definición formaliza el concepto de respuesta para los distintos tipos de consultas.

Definición 13 Sea **query** una consulta para la base de datos **db**. Su *respuesta con respecto a* **db**, a la que denotaremos como $[\![\texttt{query}]\!]_{\texttt{db}}$, se define por casos:

- consulta estándar (no hipotética): $[\![\texttt{sel\_stm}]\!]_{\texttt{db}} = [\![\texttt{sel\_stm}]\!]^{fix^{\texttt{db}}}$.
- consulta hipotética: si **R sch := sel_stm**$_{\texttt{R}}$ es la definición de **R** en **db**, entonces:
  - $[\![\texttt{assume sel\_stm}' \texttt{ in R sel\_stm}]\!]_{\texttt{db}} = [\![\texttt{sel\_stm}]\!]^{fix^{\texttt{db}'}}$, donde $\texttt{db}' = \texttt{db}[\texttt{R sch := sel\_stm}_{\texttt{R}} \texttt{ union sel\_stm}'\texttt{/ R sch := sel\_stm}_{\texttt{R}}]$.
  - $[\![\texttt{assume sel\_stm}' \texttt{ not in R sel\_stm}]\!]_{\texttt{db}} = [\![\texttt{sel\_stm}]\!]^{fix^{\texttt{db}'}}$, donde $\texttt{db}' = \texttt{db}[\texttt{R sch := sel\_stm}_{\texttt{R}} \texttt{ except sel\_stm}'\texttt{/ R sch := sel\_stm}_{\texttt{R}}]$. □

Ejemplo 18 Sea **db** la siguiente base de datos :

```
R1 (A int):= select 1 union select 2 union select 3;
R2 (A int):= select 1 union select 3 union select 5 except
             select R1.A from R1 where R1.A1 or R1.2;
R3 (A int):= select R2.A from R2 union
             select R3.A*2 from R3 where R3.A<5;
```

Para la explicación posterior, identificamos **sel_stm**$_{\texttt{R2}}$ como el **sel_stm** que define **R2**, es decir:

sel_stm$_{\texttt{R2}} \equiv$
```
select 1 union select 3 union select 5
 except
select R1.A from R1 where R1.A=1 or R1.A=2;
```

Consideremos la siguiente consulta hipotética que denotamos como **query**:

```
assume select R1.A from R1 where R1.A<3 in R2, select 3 not in R2
       select R3.A from R3
```

Entonces $[\![\texttt{query}]\!]_{\texttt{db}} = [\![\texttt{select R3.A from R3}]\!]^{fix^{\texttt{db}'}}$, donde $\texttt{db}' = (\texttt{db})\theta\sigma$ siendo:

$\theta = [\texttt{R2 := sel\_stm}'_{\texttt{R2}}\texttt{/ R2 := sel\_stm}_{\texttt{R2}}]$,

$\sigma = [\mathtt{R2} := \mathtt{sel\_stm'_{R2}}$ except select 3/$\mathtt{R2} := \mathtt{sel\_stm'_{R2}}]$,

$\mathtt{sel\_stm'_{R2}} \equiv \mathtt{sel\_stm_{R2}}$ union select R1.A from R1 where R1.A<3.

Por tanto, $\mathbf{db}'$ es la siguiente base de datos:

```
R1 (A int):= select 1 union select 2 union select 3;
R2 (A int):= ((select 1 union select 3 union select 5 except
             select R1.A from R1 where R1.A1 or R1.2) union
             select R1.A from R1 where R1.A<3) except select 3;
R3 (A int):= select R2.A from R2 union
             select R3.A*2 from R3 where R3.A<5;
```

Con este ejemplo mostramos cómo se obtiene una nueva base de datos aplicando las sustituciones correspondientes para calcular una consulta hipotética. Tal y como introducimos al principio de esta sección es necesario obtener las relaciones necesarias para calcular una consulta. En este caso las relaciones necesarias para calcular la consulta son R2 y R3. □

El cálculo de una consulta no hipotética para una base de datos no presenta mucha dificultad dado que el valor de $[\![\mathtt{sel\_stm}]\!]_{\mathbf{db}}$ es $[\![\mathtt{sel\_stm}]\!]^{fix^{\mathbf{db}}}$ y $fix^{\mathbf{db}}$ es conocido y coincide con la instancia de la base de datos. Sin embargo, el caso de la consulta hipotética **sel_hyp** es más complejo. Su significado pretendido es la interpretación de una instrucción *select* con respecto a una nueva base de datos $\mathbf{db}'$ en la cual hemos cambiado la definición de algunas relaciones. Dado que hemos incorporado las suposiciones a sus respectivas relaciones, $\mathbf{db}'$ debe ser una base de datos también estratificable para poder definir la interpretación $fix^{\mathbf{db}'}$.

Usando la semántica de punto fijo estratificada podemos simplificar algunas partes del cómputo de $fix^{\mathbf{db}'}$ (veáse también la sección 4 de [B.2]):

- En primer lugar, el grafo de dependencias $DG_{\mathbf{db}'}$ es una extensión de $DG_{\mathbf{db}}$ dado que $\mathtt{RN}_{\mathbf{db}'} = \mathtt{RN}_{\mathbf{db}}$, i.e., toda relación de $\mathbf{db}'$ estaba inicialmente en **db**. Sin embargo, debemos tener en cuenta la nueva definición de la relación R: R sch := $\mathtt{sel\_stm_R}$ (**union**|**except**) $\mathtt{sel\_stm'}$. Los arcos desde $\mathtt{sel\_stm_R}$ hacia R ya pertenecen al grafo inicial $DG_{\mathbf{db}}$.

  Podemos construir $DG_{\mathbf{db}'}$ a partir de $DG_{\mathbf{db}}$ añadiendo los nodos para todo $\mathtt{R}' \in \mathtt{RN}_{\mathtt{sel\_stm'}}$ y los siguientes arcos: para todo R′ se añade un arco desde R′ hasta R. Etiquetamos negativamente este arco en el caso de **except**, o bien si $\mathtt{R}' \in \mathtt{RN}^{\neg}_{\mathtt{sel\_stm'}}$.

  La estratificación $str' : \mathtt{RN}_{\mathbf{db}'} \to \{1, \ldots, n\}$ para $\mathbf{db}'$, si existe debe satisfacer que $str'(\mathtt{R}) \geq str(\mathtt{R})$. En general la instrucción *select* que define R en $\mathbf{db}'$ contiene la instrucción $\mathtt{sel\_stm_R}$, que también define R en **db**.

- En segundo lugar, para calcular el significado $[\![sel(\mathtt{sel\_hyp})]\!]^{fix^{\mathbf{db}'}}$, solo necesitamos calcular $fix^{\mathbf{db}'}(\mathtt{R}')$ para las relaciones de R′ tal que dicha relación en $\mathtt{RN}_{sel(\mathtt{sel\_hyp})}$ depende de R′. Además, no es necesario calcular $fix^{\mathbf{db}'}$ para el estrato 1 dado que $i = str'(\mathtt{R})$ ($i = min\{str'(\mathtt{R}_j) | 1 \leq j \leq k\}$). Por tanto $fix^{\mathbf{db}'}$ puede ser calculado a partir de $fix^{\mathbf{db}}$:

  - Si suponemos que hay asunciones en las relaciones $\mathtt{R}_1, \ldots, \mathtt{R}_k$ entonces se cumplirá que $fix^{\mathbf{db}'}(\mathtt{R}') = fix^{\mathbf{db}}(\mathtt{R}')$ para todo R′ con $str'(\mathtt{R}') < i$.
  - Si $S = \{\mathtt{R}'' \mid \mathtt{R}' \in \mathtt{RN}_{sel(\mathtt{sel\_hyp})}$ y R′ depende de $\mathtt{R}''\}$, entonces obtenemos $fix^{\mathbf{db}'}$ a partir de $fix^{\mathbf{db}}$ de la siguiente forma:
    1. Se calcula $fix_i^{\mathbf{db}'}(\mathtt{R}')$ desde $fix_{i-1}^{\mathbf{db}}$ para todas las relaciones de $\mathtt{R}' \in S$ que cumplen que $str'(\mathtt{R}') = i$.

2. Se calcula $fix_j^{\mathtt{db}'}(\mathtt{R}')$ desde $fix_{j-1}^{\mathtt{db}'}$ para las relaciones $\mathtt{R}' \in S$ y $str'(\mathtt{R}') = j$, $j = i + 1 \ \ldots \ str'(sel(\mathtt{sel\_hyp}))$.

Ejemplo 19 Consideremos la base de datos y la consulta del ejemplo 18. Sea str una estratificación para $\mathtt{db}$, tal que $str(\mathtt{R1}) = 1$, $str(\mathtt{R2}) = 2$, $str(\mathtt{R3}) = 3$. En este caso str también es una estratificación válida para $\mathtt{db}'$ y se puede verificar que:

- $fix^{\mathtt{db}}(\mathtt{R1}) = \{(1), (2), (3)\}$,
- $fix^{\mathtt{db}}(\mathtt{R2}) = \{(3), (5)\}$,
- $fix^{\mathtt{db}}(\mathtt{R3}) = \{(3), (5), (6)\}$.

Como estamos calculando $fix^{\mathtt{db}'}$, debemos tener en cuenta que $\mathtt{RN}_{sel(\mathtt{query})} = \{\mathtt{R3}\}$. Entonces $S = \{\mathtt{R}'' \mid \mathtt{R}' \in \{\mathtt{R3}\}$ y $\mathtt{R}'$ depende de $\mathtt{R}''\} = \{\mathtt{R1}, \mathtt{R2}, \mathtt{R3}\}$, el cómputo puede empezar en el estrato $2 = str(\mathtt{R2})$, siendo $fix_1^{\mathtt{db}'} = fix_1^{\mathtt{db}}$.

Ahora $\mathtt{R2}$ es la única relación en S en el estrato 2.

$$fix_2^{\mathtt{db}'}(\mathtt{R2}) = \{(1), (2), (5)\}.$$

De manera análoga procedemos para el último estrato 3 y así calcular $fix_3^{\mathtt{db}'}(\mathtt{R3})$ para obtener la respuesta:

$$fix_3^{\mathtt{db}'}(\mathtt{R3}) = \{(1), (2), (4), (5), (8)\} = [\![\mathtt{select\ R3.A\ from\ R3}]\!]^{fix^{\mathtt{db}'}} = [\![\mathtt{query}]\!]_{\mathtt{db}}. \qquad \square$$

Una vez formalizada la semántica de las consultas continuamos presentando la semántica de vistas del lenguaje *HR-SQL*.

3.3.3. La semántica de las vistas

Como aparece en la sección 3.2.2, para definir una vista asignamos un nombre de vista a una consulta. El hecho de definir vistas hipotéticas en una fase posterior a la definición de la base de datos permite hacer un análisis de los cambios demandados por las suposiciones en la base de datos y hacer un cómputo lo más eficiente posible cuando se implementa el sistema. Nótese que, dependiendo de la complejidad de la definición de la vista, el cálculo podría demandar en el caso peor la modificación de todas las relaciones en la base de datos. Sin embargo, dado que usamos una semántica de punto fijo estratificada podemos usar el grafo de dependencias para recalcular solamente las relaciones que necesitamos modificar en el contexto de una vista hipotética.

El significado de la definición de un conjunto de vistas $\mathtt{vd}$ debe establecer la correspondencia entre cada nombre y su interpretación. Sin embargo, esa interpretación debe considerar la base de datos original extendida con las nuevas definiciones que aparecen en $\mathtt{vd}$ y extender también el grafo con las dependencias demandadas por la sentencia $\mathtt{sel\_hyp}$ correspondiente. Se ha diseñado la estratificación de forma que se asigna un estrato nuevo a cada nombre de vista, lo que permite reutilizar el punto fijo almacenado de la base de datos para calcular la semántica de las vistas hipotéticas. Los detalles de esta nueva estratificación aparecen en la sección 3.4.2.

Por legibilidad formalizamos cómo se obtiene la semántica para la definición de una sola vista y terminamos generalizando cómo proceder para calcular la semántica de una secuencia de vistas $\mathtt{vd}$.

Definición 14 Definimos el significado de una vista en el contexto de una base de datos $\mathtt{db}$ diferenciando entre $\mathtt{V}$ (no hipotética) y $\mathtt{HV}$ (hipotética).

- Sea $\mathtt{V\ sch\ :=\ sel\_stm}$ la definición de una vista no hipotética para $\mathtt{db}$.

  El *significado* de $\mathtt{V}$ con respecto a $\mathtt{db}$, que denotamos como $[\![\mathtt{V}]\!]_{\mathtt{db}}$, es igual a $[\![\mathtt{sel\_stm}]\!]_{\mathtt{db}'}$, donde $\mathtt{db}'$ es el resultado de extender $\mathtt{db}$ con $\mathtt{V\ sch\ :=\ sel\_stm}$ como una nueva relación.

- Sea $\mathtt{HV\ sch\ :=\ sel\_hyp}$ la definición de una vista hipotética para $\mathtt{db}$.

  El *significado* de $\mathtt{HV}$ con respecto a $\mathtt{db}$, que denotamos como $[\![\mathtt{HV}]\!]_{\mathtt{db}}$, es igual a $[\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}'}$, donde $\mathtt{db}'$ es el resultado de extender $\mathtt{db}$ con $\mathtt{HV\ sch\ :=}\ sel(\mathtt{sel\_hyp})$ como una nueva relación. □

En $\mathtt{V\ sch\ :=\ sel\_stm}$, el valor $[\![\mathtt{V}]\!]_{\mathtt{db}} = [\![\mathtt{sel\_stm}]\!]_{\mathtt{db}'} = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}'}}$ depende del punto fijo de una nueva base de datos que debe ser estratificable. $\mathtt{db}'$ es igual a $\mathtt{db}$ extendida con $\mathtt{V\ sch\ :=\ sel\_stm}$.

La nueva $\mathtt{db}$ es no estratificable si $\mathtt{V}$ aparece en alguna instrucción $\mathtt{except}$ dentro de $\mathtt{sel\_stm}$. En otro caso, el punto fijo de la nueva base de datos es igual al punto fijo de $\mathtt{db}$ salvo para la relación $\mathtt{V}$. Nótese que $\mathtt{RN}_{\mathtt{db}'} = \mathtt{RN}_{\mathtt{db}} \cup \{\mathtt{V}\}$ si no hay nuevas dependencias en $\mathtt{db}$ producidas por $\mathtt{V}$.

Por tanto, si k es el máximo estrato de $\mathtt{db}$ (con n relaciones) entonces la estratificación str' para $\mathtt{db}'$ se puede definir como $str' : \mathtt{RN}_{\mathtt{db}'} \to \{1, \ldots, n+1\}$, con $str'(\mathtt{R}) = str(\mathtt{R})$ para todo $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$ y $str'(\mathtt{V}) = k+1$. Así, para $i = 1..k$ tendremos que $fix_i^{\mathtt{db}'} = fix_i^{\mathtt{db}}$. Y por tanto:

$$fix^{\mathtt{db}'} = fix_{k+1}^{\mathtt{db}'} = \bigsqcup_{m \geq 0} (T_{k+1}^{\mathtt{db}'})^m (fix^{\mathtt{db}}),$$

donde $fix^{\mathtt{db}'}$ es una extensión del punto fijo $fix^{\mathtt{db}}$ conocido. Esto supone que solo debemos realizar el cómputo para el último estrato $k+1$, esto es, para la relación $\mathtt{V}$:

$$[\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}'}} = fix_{k+1}^{\mathtt{db}'}(\mathtt{V}).$$

La semántica del caso $\mathtt{HV\ sch\ :=\ assume\ sel\_stm'\ [not]\ in\ R\ sel\_stm}$ requiere modificar la base de datos de dos formas:

1. $[\![\mathtt{HV}]\!]_{\mathtt{db}} = [\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}'}$, de acuerdo con la definición 14, donde $\mathtt{db}'$ es el resultado de extender $\mathtt{db}$ con $\mathtt{HV\ sch\ :=\ sel\_stm}$.
2. $[\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}'} = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}''}}$, de acuerdo con la definición 13, donde $\mathtt{db}'' = \mathtt{db}'[\mathtt{R\ sch\ :=\ sel\_stm}_{\mathtt{R}}\ (\mathtt{union} \mid \mathtt{except})\ \mathtt{sel\_stm}'/\mathtt{R\ sch\ :=\ sel\_stm}_{\mathtt{R}}]$.

Con el punto 1 extendemos la base de datos con una nueva relación $\mathtt{HV}$. La nueva definición de $\mathtt{HV}$ sin su hipótesis ($\mathtt{HV\ sch\ :=\ sel\_stm}$) es sintácticamente correcta. Sin embargo, $\mathtt{HV\ sch\ :=\ sel\_hyp}$ no se permitiría como una definición válida en la base de datos original.

Con el punto 2 la suposición se incorpora a la relación correspondiente tal y como se explica en la sección 3.3.2. De esta forma, las nuevas definiciones de relación en $\mathtt{db}''$ son:

$$\begin{gathered}\mathtt{HV\ sch\ :=}\ sel(\mathtt{sel\_hyp});\\ \mathtt{R\ sch\ :=\ sel\_stm}_{\mathtt{R}}\ (\mathtt{union} \mid \mathtt{except})\ \mathtt{sel\_stm}';\end{gathered}$$

El cálculo de $fix_{\mathtt{db}''}$ se puede simplificar. En primer lugar el grafo de dependencias $DG_{\mathtt{db}''}$ se crea a partir de $DG_{\mathtt{db}}$ añadiendo nuevos arcos hacia la relación $\mathtt{R}$. Es decir, añadimos un nuevo nodo para la vista $\mathtt{HV}$ con sus arcos correspondientes. Para todo $\mathtt{R}' \in \mathtt{RN}_{sel(\mathtt{sel\_hyp})}$ hay un arco desde $\mathtt{R}'$ hasta $\mathtt{HV}$ que se etiqueta negativamente si $\mathtt{R}' \in \mathtt{RN}^{\neg}_{sel(\mathtt{sel\_hyp})}$.

Una estratificación $str' : \mathtt{RN}_{\mathtt{db}''} \to \{1, \ldots, n+1\}$ de $\mathtt{db''}$ si existe, puede asignar el estrato $k+1$ a $\mathtt{HV}$ tal y como sucede en el caso no hipotético, siendo k el último estrato de la $\mathtt{db}$ original.

Así, $fix^{\mathtt{db}''}_{k}$ se puede calcular (siguiendo las ideas de la sección 3.3.2) partiendo del punto fijo almacenado para la base de datos con sus correspondientes reemplazamientos. En este caso el cálculo de $fix^{\mathtt{db}''}_{k+1}$ considerará solo $\mathtt{HV}$. Para el estrato $k+1$ se cumple que:

$$fix^{\mathtt{db}''}_{k+1}(\mathtt{HV}) = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}''}}.$$

Ejemplo 20 Consideramos la base de datos $\mathtt{db}$ del ejemplo 18 y la vista hipotética:

```
HV (A int) := assume select R1.A from R1 where R1.A<3 in R2
                     select 3 not in R2
              select R3.A from R3 union
              select HV.A*3 from HV where HV.A<3;
```

Siguiendo la definición 14:

$$[\![\mathtt{HV}]\!]_{\mathtt{db}} = [\![\mathtt{select\ R3.A\ from\ R3\ union\ select\ HV.A} * 3\ \mathtt{from\ HV\ whereHV.A} < 3]\!]^{fix^{\mathtt{db}''}},$$

donde $\mathtt{db}''$ es una extensión de la base de datos $\mathtt{db}'$ del ejemplo 18 con:

$$\mathtt{HV(A\ int)} := \mathtt{select\ R3.A\ from\ R3\ union\ select\ HV.A} * 3\ \mathtt{from\ HV\ where\ HV.A} < 3;$$

Una función str' que aumenta str de forma que $str'(\mathtt{HV}) = 4$ es una estratificación válida para la nueva base de datos (que tiene en cuenta las definiciones extendidas con las suposiciones). Para $1 \leq i \leq 3$ se cumple que $fix^{\mathtt{db}''}_{i} = fix^{\mathtt{db}'}_{i}$ tal y como sucedía en el ejemplo 19. Dado que $[\![\mathtt{HV}]\!]_{\mathtt{db}}$ coincide con $fix^{\mathtt{db}''}(\mathtt{HV})$, solo necesitamos calcular:

$$fix^{\mathtt{db}''}_{4}(\mathtt{HV}) = (\bigsqcup_{m \geq 0} (T^{\mathtt{db}''}_{4})^{m}(fix^{\mathtt{db}''}_{3}))(\mathtt{HV}) = \{(1), (2), (3), (4), (5), (6), (8)\}. \qquad \square$$

Semántica de las secuencias de vistas simultáneas

La idea es que la semántica de $\mathtt{vd}$ asocia a cada nombre de vista en $\mathtt{vd}$ la interpretación de la consulta $\mathtt{sel\_stm}$ que la define. Sin embargo, si hay más de una vista no hipotética en $\mathtt{vd}$, imponemos una restricción adicional: no podemos asignar la semántica $[\![\mathtt{V}]\!]_{\mathtt{db}}$ a $[\![\mathtt{sel\_stm}]\!]_{\mathtt{db}'}$ dado que $\mathtt{db}'$ es el resultado de extender $\mathtt{db}$ con $\mathtt{V\ sch\ :=\ sel\_stm}$. Esto se debe a que otros nombres definidos en $\mathtt{vd}$ distintos de $\mathtt{V}$ pueden aparecer dentro de $\mathtt{sel\_stm}$ sin estar definidos en $\mathtt{db}'$. A continuación definimos la semántica de una secuencia de vistas $\mathtt{vd}$.

Definición 15 Sea $\mathtt{db}$ una base de datos y sea $\mathtt{vd}$ una definición de vistas para $\mathtt{db}$ que denotamos con la siguiente secuencia:

$$\begin{array}{lll}
\mathtt{V}_1\ \mathtt{sch}_1 & ::= & \mathtt{sel_stm}_1; \\
 & \ldots & \\
\mathtt{V}_m\ \mathtt{sch}_m & ::= & \mathtt{sel_stm}_m; \\
\mathtt{HV}_1\ \mathtt{sch}_1 & ::= & \mathtt{sel_hyp}_1; \\
 & \ldots & \\
\mathtt{HV}_r\ \mathtt{sch}_r & ::= & \mathtt{sel_hyp}_r;
\end{array}$$

Definimos la semántica de $\mathtt{vd}$ como una función que asocia $[\![\mathtt{V}_j]\!]_{\mathtt{db}'}$ a $\mathtt{V}_j$ para $j = 1..m$ y $[\![\mathtt{HV}_j]\!]_{\mathtt{db}'}$ a $\mathtt{HV}_j$ para $j = 1..r$, donde $\mathtt{db}'$ es el resultado de extender $\mathtt{db}$ con:

$$\mathtt{V}_1\ \mathtt{sch}_1{:=}\mathtt{sel\_stm}_1;\ \ldots;\ \mathtt{V}_m\ \mathtt{sch}_m{:=}\mathtt{sel\_stm}_m; \qquad \square$$

Nótese que, de acuerdo con la definición 14, $[\![\mathtt{V}_j]\!]_{\mathtt{db}'} = [\![\mathtt{sel\_stm}_j]\!]_{\mathtt{db}''}$ para todo $j = 1..m$, donde $\mathtt{db}''$ es el resultado de extender $\mathtt{db}'$ con $\mathtt{V}_j\ \mathtt{sch}_j{:=}\ \mathtt{sel\_stm}_j$. Sin embargo, esta definición ya aparece en $\mathtt{db}'$ y por tanto se tiene que $\mathtt{db}'' = \mathtt{db}'$.

$\mathtt{HV}_1, \ldots, \mathtt{HV}_r$ no debe aparecer en $\mathtt{sel\_stm}_j$ porque sus definiciones no son necesarias en $\mathtt{db}'$. No obstante, para todo $1 \leq j \leq r$ se cumple $[\![\mathtt{HV}_j]\!]_{\mathtt{db}'} = [\![\mathtt{sel\_hyp}_j]\!]_{\mathtt{db}''}$ donde $\mathtt{db}''$ es el resultado de extender $\mathtt{db}'$ con $\mathtt{HV}_j\ \mathtt{sch}_j\ \mathtt{:=}\ \mathit{sel}(\mathtt{sel\_hyp}_j)$. Este hecho permite que la definición de $\mathtt{HV}_j$ sea recursiva.

Para calcular la semántica de todas las vistas en una definición simultánea, las vistas hipotéticas deben ser calculadas de una en una. Además debemos tener en cuenta las siguientes consideraciones:

- Como sucedía en el caso de un sola vista, $\mathtt{db}'$ debe ser también estratificable. Si $\mathtt{db}'$ es estratificable podemos encontrar una estratificación str' para $\mathtt{db}'$ que cumpla que $\mathit{str}'(\mathtt{V}_j) > n$ para todo $1 \leq j \leq m$.
- La interpretación $\mathit{fix}^{\mathtt{db}'}$ se puede obtener estrato por estrato:
  - Empezando desde $\mathit{fix}^{\mathtt{db}}$ para el caso no hipotético.
  - Para el caso hipotético se puede comenzar con cada vista hipotética de forma independiente partiendo siempre de $\mathit{fix}^{\mathtt{db}'}$ como la interpretación inicial y procesando cada vista como en el caso de una sola vista. El procesamiento debe ser repetido tantas veces como vistas hipotéticas haya en la secuencia $\mathtt{vd}$.

Se han implementado dos instancias del marco teórico: en PostgreSQL y DB2. En la siguiente sección presentamos los sistemas *R-SQL* y *HR-SQL*, su forma de funcionamiento y algunos ejemplos de cómputo del punto fijo para sus dos implementaciones.

3.4. El sistema *R-SQL*

En esta sección presentamos el funcionamiento del sistema *R-SQL* que se describe en las publicaciones [B.1,B.3]. El sistema está implementado en SWI-Prolog y utiliza el sistema de código abierto PostgreSQL como SGBDR. El punto fijo para las bases de datos se calcula siguiendo la semántica operacional que presentamos en la sección 3.3.

Procesamiento de las bases de datos

Antes de procesar una base de datos en *R-SQL*, debemos cargar los archivos fuente del sistema en Prolog mediante la siguiente instrucción:

```
:-[rsql].
```

Una vez que el sistema está cargado, el usuario puede procesar la definición de una base de datos `dbDef` con el comando `process(dbDef)`. Entonces el sistema analiza sintácticamente la definición de la base de datos y después calcula el grafo de dependencias y la estratificación

en caso de que exista (si no existe se lanza un mensaje de error y la ejecución termina). Finalmente el sistema genera un *script* Python que se encarga de automatizar la conexión con PostgreSQL y que materializa las relaciones en este SGBDR. Tras este proceso el usuario se puede conectar a PostgreSQL para hacer consultas y modificar las relaciones de la base de datos. En esta memoria presentamos solo la implementación de *R-SQL* sobre PostgreSQL aunque también se ha implementado sobre MySQL.

El algoritmo de cómputo de punto fijo del sistema se explica en la sección 3.1 de [B.3], y es muy similar al que presentamos en la sección 3.5.2 para *HR-SQL*. A continuación proponemos un ejemplo de cálculo del punto fijo de una base de datos y mostramos el código Python que se genera. Terminamos la sección presentando otra de las aportaciones que provienen de [B.3]: una optimización en el cálculo de la estratificación que minimiza el número de relaciones en cada estrato para mejorar la eficiencia del cómputo.

3.4.1. Cómputo de las bases de datos *R-SQL*

En el siguiente ejemplo, extraído de [B.3], presentamos el cálculo del punto fijo de una base de datos *R-SQL* paso a paso. Por legibilidad utilizamos mayúsculas para representar palabras reservadas de SQL cuando pertenecen a un *script* de Python y minúsculas cuando forman parte de una definición *R-SQL*. Sin embargo, el sistema admite las palabras reservadas de SQL tanto en mayúsculas como en minúsculas al igual que la mayoría de los SGBDR.

Ejemplo 21 Se trata de un ejemplo de cierre transitivo sobre una base de datos de vuelos entre distintas ciudades similar al ejemplo 17. Las ciudades de origen y destino de este ejemplo son Lisboa, Madrid, París, Londres y Nueva York, y se representan respectivamente con las constantes `lis`, `mad`, `par`, `lon`, `ny`.

La relación `reach` consiste en los posibles vuelos entre estas ciudades, que pueden concatenar (o no) varios vuelos. La relación `travel` es similar pero también nos devuelve la duración de los viajes en `time`.

```
flight(frm varchar(10), to varchar(10), time float) :=
    select 'lis', 'mad', 1.0 union
    select 'mad', 'par', 1.5 union
    select 'par', 'lon', 2.0 union
    select 'lon', 'ny',  7.0 union
    select 'par', 'ny',  8.0;

reach(frm varchar(10), to varchar(10)) :=
    select flight.frm, flight.to from flight union
    select reach.frm,  flight.to from reach,flight
      where reach.to = flight.frm;

travel(frm varchar(10), to varchar(10), time float) :=
    select flight.frm, flight.to, flight.time from flight union
    select flight.frm, travel.to, flight.time+travel.time
      from flight, travel where flight.to = travel.frm;
```

Nótese que si el cierre transitivo de `flight` tiene un ciclo, la relación `travel` puede ser infinita porque al computarla se irian sumando tiempos de forma no terminante al recorrer

sucesivamente el ciclo. Para resolver este problema deberíamos imponer una limitación en su definición para asegurar la terminación en el cálculo de su resultado (como puede ser añadir un tiempo máximo). La situación descrita es un problema que surge en cualquier base de datos relacional cuando se trata con ciclos en definiciones de grafos.

Sin embargo, esta limitación no afecta a la relación **reach** que se puede calcular de forma finita en el sistema *R-SQL* y llevaría a un cómputo no terminante en otros SGBDR. Esto se debe a que el sistema calcula el punto fijo asegurando la terminación para estos casos en los que se permite la existencia de ciclos en grafos dirigidos dado que se comprueba si se han añadido nuevas tuplas en cada iteración del bucle. En la sección 3.5.2 podemos ver cómo se genera el código del *script*.

Seguimos presentando cómo se definen las relaciones del ejemplo propuesto en *R-SQL*. La relación **madAirport** contiene los vuelos que salen o aterrizan en Madrid, mientras que **avoidMad** contiene aquellos vuelos que ni aterrizan ni salen de Madrid.

```
madAirport(frm varchar(10), to varchar(10)) :=
    select reach.frm, reach.to from reach
     where (reach.frm = 'mad' or reach.to = 'mad');

avoidMad(frm varchar(10), to varchar(10)) :=
    select reach.frm, reach.to from reach except madAirport;
```

Esta combinación de la instrucción **except** y recursión no se permite en el estándar SQL-99 como se muestra en [33].

El grafo de dependencias para las relaciones de esta base de datos aparece en la figura 3.6. Como es habitual anotamos con el símbolo ¬ las dependencias negativas del grafo.

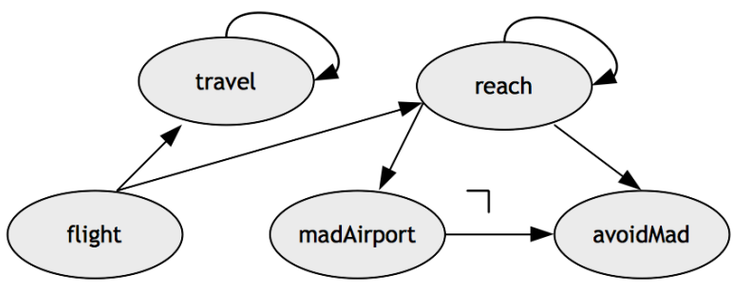


Figura 3.6: DG_{db} del ejemplo 21.

A continuación mostramos el código Python que genera *R-SQL* para materializar las relaciones en tablas del SGBDR. Hemos utilizado la biblioteca *psycopg2* que permite formular una consulta a PostgreSQL con la instrucción:

cursor.execute(" *query*")

donde *query* es una consulta SQL válida. Como Python no proporciona en su sintaxis bucles **repeat** (o **do-while**), hemos implementado una construcción similar mediante **while True** y la sentencia **break** cuando se cumple la condición de salida.

En el *script* generado se crean en primer lugar las tablas:

```
  cursor.execute("CREATE TABLE flight
                    (frm varchar(10), to varchar(10), time float);")
  cursor.execute("CREATE TABLE travel
                    (frm varchar(10), to varchar(10), time float);")
```

y para el estrato 1 se genera el siguiente código:

```
# Código para el estrato 1
# Fragmento out
cursor.execute("INSERT INTO flight
                (SELECT 'lis','mad',1   UNION
                 SELECT 'mad','par',1.5 UNION
                 SELECT 'par','lon',2   UNION
                 SELECT 'lon','ny',7    UNION
                 SELECT 'par','ny',8)")
```

Para la generación de código SQL se utiliza un algoritmo muy similar al presentado en la sección 3.5.2. Además se aplica la partición de definiciones para mejorar la eficiencia que aparece también en esta sección y que denominamos *algoritmo in/out*. Lo que se pretende con este algoritmo es reducir el número de llamadas (e inserciones) al SGBDR dentro del bucle y con ello mejorar el rendimiento del sistema. Se trata de una técnica que se usa habitualmente en sistemas de BDD [122].

El estrato 2 contiene la relación `travel` cuya definición se divide en dos partes: la recursiva (*in*) que estará dentro del bucle **while** y la no-recursiva (*out*) que se compone de tuplas que que no se calculan recursivamente y no es necesario incluir en el bucle iterativo. Mostramos ambos fragmentos:

```
# Código generado para el estrato 2
# Fragmento out
cursor.execute("INSERT INTO travel (SELECT * FROM flight);")

# Fragmento in
while True:
  cursor.execute("INSERT INTO travel
                 (SELECT flight.frm,travel.to,flight.time+travel.time
                   FROM flight,travel
                   WHERE flight.to = travel.frm)
                  EXCEPT
                  SELECT * FROM travel;")

  newSize = relSize(["travel"])
  if (newSize != size):
      size = newSize
  else:
      break
```

donde la función `relSize(<list of relations>)` devuelve el número de tuplas para una relación dada. Las tuplas añadidas para `travel` en cada iteración se muestran en la siguiente tabla:

	Conjunto de tuplas insertadas
Fragmento *out*	{(lon, ny, 7,0), (par, lon, 2,0), (par, ny, 8,0), (mad, par, 1,5), (lis, mad, 1,0)}
Fragmento *in*: iteración 1	{(lis, par, 2,5), (par, ny, 9,0), (mad, ny, 9,5), (mad, lon, 3,5)}
Fragmento *in*: iteración 2	{(lis, ny, 10,5), (lis, lon, 4,5), (mad, ny, 10,5)}
Fragmento *in*: iteración 3	{(lis, lon, 4,5), (mad, ny, 10,5), (lis, ny, 11,5)}

De manera análoga el sistema produce el código Python para los estratos 3 y 4, que corresponden respectivamente a las relaciones `reach` y `madAirport`. Para concluir, presentamos el código del *script* para generar la relación `avoidMad` que completa el estrato número 5:

```
# Código generado para el estrato 5
# Fragmento out
cursor.execute("INSERT INTO avoidMad
                (SELECT travel.frm,travel.to FROM travel
                 EXCEPT SELECT * FROM madAirport)");
```

Con este fragmento completamos el código que tiene como objetivo el cálculo del el punto fijo para la base de datos propuesta. Los valores de `flight`, `madAirport` y `avoidMad` se representan gráficamente en la figura 3.7. □

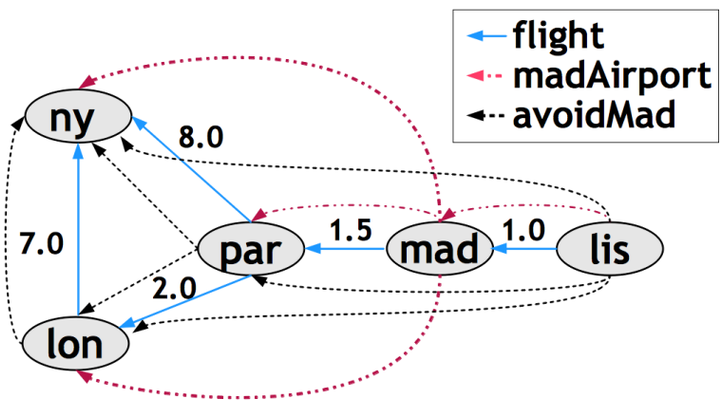


Figura 3.7: Representación gráfica las tuplas de la base de datos del ejemplo 21.

Una vez que la base de datos *R-SQL* ha sido procesada, las tablas resultantes están disponibles para ser consultadas en PostgreSQL. El usuario puede formular consultas o bien directamente en PostgreSQL, o bien desde el inductor de comandos del sistema *R-SQL*. En ambos casos sin ningún cómputo adicional del punto fijo.

Terminamos esta sección presentando la propuesta de estratificación del sistema *R-SQL* que aparece en la sección 3.5 de [B.3] y que mejora la versión del sistema presentada en [B.1]. Está optimización también se aplica al sistema *HR-SQL* que presentamos en la última sección del capítulo.

3.4.2. El algoritmo de estratificación

El algoritmo que presentamos a continuación trata de minimizar el número de relaciones en cada estrato para mejorar el cómputo del punto fijo que aparece en la sección 3.5.2.

El objetivo es que cada estrato i contenga o bien una sola relación, o bien un conjunto de relaciones que sean mutuamente recursivas. Por tanto, no es necesario iterar los bucles tantas veces como demande la relación que requiera mayor número de iteraciones de cada estrato. Esta mejora también se puede aplicar a la estratificación de las bases de datos del sistema $HH\neg(\mathcal{C})$ que presentamos en el capítulo anterior.

Presentaremos el nuevo algoritmo de estratificación mediante un grafo de dependencias. Para el ejemplo de la figura 3.8 una estratificación correcta es aquélla que asigna el estrato 1 a las relaciones $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ y el estrato 2 a las relaciones $\{\mathtt{f}, \mathtt{g}\}$. De hecho, ésta es la estratificación que asigna el sistema $HH\neg(\mathcal{C})$ a este grafo.

Sin embargo, en la figura 3.8 podemos ver que solamente es necesario que las relaciones mutuamente recursivas $\mathtt{b}$ y $\mathtt{c}$ pertenezcan al mismo estrato (debido a la dependencia mutua entre ambas relaciones).

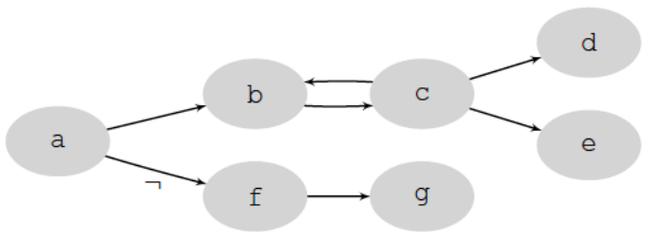


Figura 3.8: Ejemplo de grafo de dependencias

Para aislar las relaciones no mutuamente recursivas en un solo estrato se calcula una estratificación que aumenta el número de estratos como vemos a continuación. Partimos de base de datos $\mathtt{db}$ y su grafo de dependencias asociado DG_{db}, el algoritmo de estratificación:

1. Calcula las componentes fuertemente conexas C de DG_{db}. En principio no tiene en cuenta las dependencias negativas. Sin embargo, una vez que se han obtenido estas componentes, debemos comprobar si existe un ciclo con alguna dependencia negativa. En ese caso $\mathtt{db}$ no es estratificable y el cómputo debe terminar. Para las dependencias de la figura 3.8 las componentes fuertemente conexas son:

   $$\{\mathtt{a}\}, \{\mathtt{f}\}, \{\mathtt{g}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{d}\} \text{ y } \{\mathtt{e}\}.$$

2. Agrupa las componentes fuertemente conexas de forma que se obtiene un nuevo grafo compuesto por nodos para cada componente C y arcos que unen dicha C con otras componentes C'. Si C contiene solamente una relación $\mathtt{R}$ y C' contiene solo la relación $\mathtt{R}'$ es inmediato que el arco correspondiente irá desde $\mathtt{R}$ hasta $\mathtt{R}'$ en DG_{db}.

   En nuestro ejemplo, las componentes $\{\mathtt{b}, \mathtt{c}\}$ se asocian al nodo $\mathtt{bc}$, mientras que el resto se asocian trivialmente con el nodo que contiene su relación correspondiente. El grafo resultante contiene los siguientes arcos:

   $$\{\mathtt{a} \to \mathtt{bc}, \mathtt{bc} \to \mathtt{d}, \mathtt{bc} \to \mathtt{e}, \mathtt{a} \to \mathtt{f}, \mathtt{f} \to \mathtt{g}\}.$$

3. El algoritmo obtiene una ordenación topológica del nuevo grafo. En el ejemplo esta ordenación es $\mathtt{a} < \mathtt{f} < \mathtt{g} < \mathtt{bc} < \mathtt{e} < \mathtt{d}$.

4. Finalmente se vuelven a separar los nodos de las componentes para obtener la ordenación topológica de las componentes fuertemente conexas y se enumeran los nodos en orden ascendente. Para nuestro ejemplo obtenemos $\{\mathtt{a}\} < \{\mathtt{f}\} < \{\mathtt{g}\} < \{\mathtt{b}, \mathtt{c}\} < \{\mathtt{e}\} < \{\mathtt{d}\}$.

   La estratificación final es:

$$str(\mathtt{a}) = 1;\ str(\mathtt{f}) = 2;\ str(\mathtt{g}) = 3;\ str(\mathtt{b}) = str(\mathtt{c}) = 4;\ str(\mathtt{e}) = 5;\ str(\mathtt{d}) = 6.$$

Ejemplo 22 Para el grafo de dependencias de la figura 3.6, el sistema *R-SQL* obtiene la siguiente estratificación:

{(1, **flight**), (2, **travel**), (3, **reach**), (4, **madAirport**), (5, **avoidMad**)}. □

3.5. El sistema *HR-SQL*

En esta sección presentamos la implementación de *HR-SQL* sobre el SGBDR DB2 que aparece en [B.2]. Este sistema puede procesar las mismas bases de datos que *R-SQL* siguiendo un algoritmo similar para generar los *scripts*.

Como hemos visto en la sección 3.3.3 se incorpora la capacidad de que el usuario pueda añadir suposiciones cuando se formulan consultas y se definen vistas. Esto aumenta la complejidad del sistema dado que estas suposiciones afectan a otras relaciones de la base de datos cuya información puede aumentar o disminuir para obtener la respuesta de una consulta. En esta sección presentamos la estructura del sistema, el algoritmo para generar *scripts* para bases de datos, consultas y vistas, y una mejora de eficiencia en la ejecución de dichos *scripts* mediante la partición de las definiciones recursivas en dos fragmentos: su caso base (*out*) y su caso recursivo (*in*). Terminaremos la sección explicando cómo *HR-SQL* calcula la semántica de vistas y consultas hipotéticas y presentando algunos resultados de eficiencia.

3.5.1. Estructura del sistema

La estructura del sistema aparece en la figura 3.9. La interfaz de usuario consiste en el siguiente inductor de comandos:

hr-db2=>

que funciona como una extensión del intérprete de comandos de DB2. El usuario puede formular cualquier entrada válida de DB2 (entrada etiquetada como A en la figura 3.9), así como consultas y comandos *HR-SQL* (etiqueta B en la figura 3.9). En el inductor de comandos se puede escribir:

- **load_db** <*db_file*>. Carga una definición de base de datos *HR-SQL* desde el archivo *db_file* y calcula el punto fijo. Las tuplas resultantes de cada relación se almacenan como tablas de DB2.
- **load_vd** <*vd_file*>. Carga una definición de vistas *HR-SQL* desde el archivo *vd_file*, calcula las tuplas de estas vistas y las materializa también en DB2.
- Una consulta hipotética (**sel_hyp**). En este caso el sistema reconoce la consulta identificando la palabra reservada **assume**.

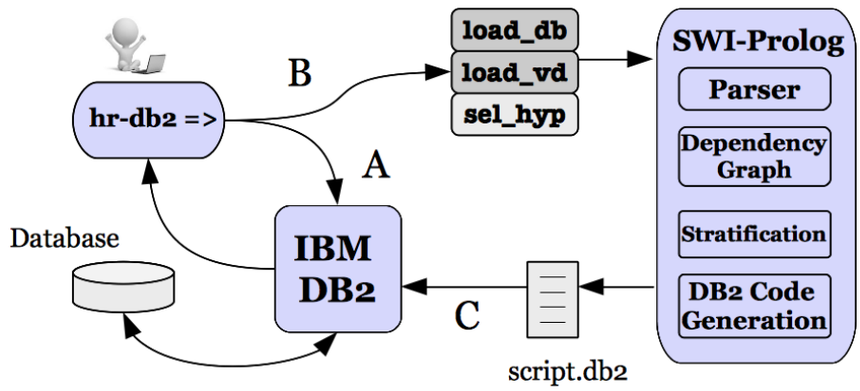


Figura 3.9: Estructura del sistema *HR-SQL*.

Estas nuevas consultas son procesadas por el sistema como se muestra en la figura 3.9. Se comienza con el análisis sintáctico (que denominamos *parser* acogiéndonos a la nomenclatura inglesa) y después se construye el grafo de dependencias y se calcula la estratificación en caso de que exista. En otro caso se lanza un mensaje de error.

Para el cálculo de la estratificación, el sistema *HR-SQL* sigue el algoritmo presentado en 3.4.2. Tras el cálculo de la estratificación, generamos automáticamente un *script SQL PL* como explicamos en la sección 3.5.2. La salida se ejecuta en el sistema de bases de datos DB2 (etiqueta C en la figura 3.9). La implementación de las vistas hipotéticas se explica en la sección 3.5.3.

Tanto el sistema (en su versión actual sobre PostgreSQL) como los ejemplos presentados en este capítulo se pueden descargar de:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/HR-SQLplus`.

3.5.2. Cálculo del punto fijo

En la figura 3.10 mostramos el algoritmo que calcula el punto fijo para una base de datos *HR-SQL*. Este algoritmo produce las instrucciones *select* necesarias (además de las sentencias `CREATE` e `INSERT`).

Esta versión del algoritmo (que se presenta en [B.2,B.3]) es una versión mejorada sobre la que se presenta en [B.1] porque simplifica el cómputo del bucle al incorporar las funciones *in* y *out* como explicamos más adelante en esta sección.

Nuestro algoritmo parte de una estratificación para la base de datos donde *numStr* es el número de estratos y $\mathtt{NR}_i$ es el conjunto de relaciones que pertenecen al estrato i.

- En primer lugar se crea una tabla para cada definición de relación de la base de datos de la forma $\mathtt{R\ sch := sel\_stm}_{\mathtt{R}}$ (línea 1).
- Después, el bucle **while** (líneas 3-10) calcula los puntos fijos:

  $$fix_1^{\mathtt{db}}, fix_2^{\mathtt{db}}, \ldots, fix_{numStr}^{\mathtt{db}}$$

  de forma iterativa. Este paso sería equivalente a iterar el operador correspondiente $T_i^{\mathtt{db}}$ que hemos presentado en la definición 11.
- La iteración n-ésima del bucle **repeat** (líneas 5-9) calcula $(T_i^{\mathtt{db}})^n(fix_{i-1})$.

- Iteramos el bucle externo mientras haya nuevas tuplas que añadir a las tablas del estrato actual. Esta comprobación se hace mediante la variable **size** que se incrementa solo si se han añadido tuplas nuevas al estrato i.

```
 1  for all R ∈ RN_db do CREATE TABLE R sch;
 2  i := 1
 3  while i ≤ numStr do
 4    for all R ∈ RN_i do INSERT INTO R out(sel_stm_R);
 5    repeat
 6      size := rel_size (RN_i)
 7      for all R ∈ RN_i do
 8        INSERT INTO R in(sel_stm_R) EXCEPT SELECT * FROM R;
 9    until size = rel_size(RN_i)
10    i := i + 1
```

Figura 3.10: Algoritmo de cálculo del punto fijo

El algoritmo *in/out*

Una de las características que mejoran la eficiencia del cálculo de relaciones con respecto al presentado en [B.1] y que aparece en [B.2] es el uso del algoritmo *in/out* para reducir el número de iteraciones dentro del bucle **while**.

La idea es que la iteración del operador $T_i^{\mathtt{db}}$ es necesaria solo para calcular la parte recursiva dentro de una instrucción `sel_stm` y los casos base de la definición se pueden sacar del bucle. Así, las funciones *in* y *out* dividen cada `sel_stm` en dos partes:

- La parte recursiva que añadimos en la instrucción `INSERT` dentro del bucle (línea 8 de la figura 3.10) y
- los casos base de la definición recursiva que extraemos del bucle (línea 4).

Podemos distinguir fácilmente los fragmentos *in* y *out* de una instrucción `sel_stm` usando el estrato de las relaciones que aparecen dentro de esta instrucción. Como hemos señalado antes, hemos diseñado la estratificación de forma que:

- Si una relación R de un estrato i depende de otra relación R′ (y no son mutuamente recursivas), se debe cumplir que el estrato de esta R′ es inferior al de i y dicha relación ha debido de ser calculada previamente.
- En otro caso, el estrato de las dos es el mismo i (en caso de ser las dos relaciones mutuamente recursivas) y ambas R y R′ se deben calcular a la vez.

Ejemplo 23 Si tenemos una relación de la forma:

$$\mathtt{R} := \mathtt{sel\_stm}_1 \ \mathtt{union} \ \mathtt{sel\_stm}_2,$$

y además $str(\mathtt{R}) = i$ y $str(\mathtt{sel\_stm}_1) < i$, entonces $\mathtt{sel\_stm}_1$ será parte del fragmento *out* pues las relaciones de las que depende están totalmente calculadas cuando se calcula fix_i y sus tuplas correspondientes se pueden insertar fuera del bucle, dado que las relaciones implicadas ya han sido calculadas en estratos anteriores y, por tanto, sabemos que no van a cambiar.

□

Las funciones in y out se definen recursivamente sobre la estructura de **sel_stm**. Por ejemplo, si **sel_stm** $\equiv$ **sel_stm**$_1$ **except** **sel_stm**$_2$ y $str($**sel_stm**$) = i$, entonces tenemos que $str($**sel_stm**$_2) < i$, y por tanto:

- $in($**sel_stm**$) = in($**sel_stm**$_1)$ **except** **sel_stm**$_2$;
- $out($**sel_stm**$) = out($**sel_stm**$_1)$ **except** **sel_stm**$_2$.

El algoritmo *in/out* aparece íntegro en la sección 3.2 de [B.3]. A continuación mostramos cómo se calculan las vistas hipotéticas para el sistema *HR-SQL* presentado en [B.3] mediante un ejemplo paso a paso.

3.5.3. Vistas y consultas en *HR-SQL*

El *script* SQL PL generado para procesar vistas sigue las ideas de la sección 3.3.3. Vamos a ilustrar mediante la vista **reachable** del ejemplo 17 los pasos que lleva a cabo el sistema para calcular las tuplas correspondientes a una vista hipotética. En este ejemplo tratamos de mostrar el cómputo para una definición recursiva que contiene suposiciones positivas y negativas una vez que la base de datos está ya calculada. Se recuerda a continuación la definición de **reachable**:

```
reachable(ori varchar(10),des varchar(10)) :=
 assume (select * from bus where bus.ori = 'VDE' union
         select * from flight not in link),
         select 'RES','SPC',1.5 in boat
 select link.ori, link.des from link union
 select link.ori, reachable.des from link, reachable
         where link.des = reachable.ori
```

En primer lugar, el sistema extiende el grafo de dependencias original con nuevos arcos debido a las suposiciones hipotéticas: dos arcos etiquetados negativamente a **link**, uno desde **bus** y otro desde **flight**. Debido a que el sistema maximiza el número de estratos, la estratificación calculada de la base de datos original es válida para la base de datos extendida.

Siguiendo las explicaciones de la sección 3.3.2, el sistema busca las relaciones que deben ser recalculadas (que hemos denominado *necesarias* en la presentación de la semántica). En concreto, para obtener las tuplas de **reachable** las relaciones necesarias son **boat** y **link**.

El algoritmo que genera las instrucciones SQL para calcular la semántica de estas relaciones junto con la nueva vista es muy similar al que hemos presentado en la figura 3.10 para calcular el punto fijo de toda la base de datos. Explicaremos las diferencias en el ejemplo.

Las relaciones necesarias para calcular la vista se crean localmente y se recalculan usando tablas temporales. La ventaja del uso de tablas temporales en un SGBDR es que permiten cómputos locales mejorando el rendimiento en memoria. Además resultan adecuadas para el cálculo de vistas con hipótesis dado que una vez que el cómputo ha finalizado son descartadas y no se materializan en el SGBDR.

Para calcular el significado de **reachable** debemos reconstruir el punto fijo para el estrato $i = min\{str($**boat**$), str($**link**$)\}$. Así creamos tablas temporales para las nuevas definiciones de **boat** y **link** que incorporan asunciones

A continuación presentamos el código SQL PL que se genera para las vistas de las nuevas definiciones (temporales) de **boat** y **link**.

En primer lugar se declaran las tablas temporales mediante las siguientes instrucciones:

```
DECLARE GLOBAL TEMPORARY TABLE link LIKE link;
DECLARE GLOBAL TEMPORARY TABLE boat LIKE boat;
```

y luego se insertan las tuplas correspondientes a sus nuevas definiciones:

```
INSERT INTO SESSION.boat
 ((SELECT 'TFS','GMZ',1)   UNION
  (SELECT 'GMZ','VDE',1.5) UNION
  (SELECT 'SPC','TFN',2 )  UNION
  (SELECT 'RES','SPC',1.5));

INSERT INTO SESSION.link
 ((SELECT * FROM flight       UNION
   SELECT * FROM SESSION.boat UNION
   SELECT * FROM bus)
    EXCEPT
  (SELECT * FROM bus WHERE bus.ori = 'VDE' UNION
   SELECT * FROM flight));
```

Podemos distinguir las tablas temporales en el código porque están prefijadas con la palabra reservada `SESSION`. Para calcular una vista hipotética procedemos de la siguiente forma:

- Partimos de la definición de la vista hipotética $\texttt{HV} := \texttt{sel\_hyp}_{\texttt{HV}}$.
- Para que la vista sea reconocida por el lenguaje SQL que procesa el SGBDR subyacente debemos eliminar las hipótesis y utilizar la una nueva definición de la vista `HV := sel_stm`, donde `sel_stm` es el resultado de remplazar `R` por `SESSION.R` dentro de $sel(\texttt{sel\_hyp}_{\texttt{HV}})$ para aquellas relaciones definidas como necesarias, i.e., `link` y `boat`.

Las tuplas de `boat` y `link` son calculadas y almacenadas en tablas temporales que son descartadas tras el cómputo. El contenido final de la tabla `reachable` aparece en la siguiente lista[1]:

ORI	DES	ORI	DES
TNF	LC	GC	MP
LC	GOM	LP	TNF
VAL	RES	GOM	VAL
TNF	GOM	LP	LC
GOM	RES	LC	VAL
LP	GOM	TNF	VAL
LC	RES	TNF	RES
LP	VAL	LP	RES

que recordamos se corresponde con los trayectos que se pueden hacer en el archipiélago canario en caso de la erupción volcánica.

[1] El resultado del sistema se presenta mediante dos columnas (ORI y DES). En esta presentación utilizamos cuatro columnas por legibilidad.

Consultas en el sistema *HR-SQL*

El proceso necesario para obtener las tuplas asociadas a las consultas hipotéticas es muy similar al explicado para las vistas. El sistema crea las tablas temporales siguiendo los mismos pasos que sigue para las vistas. Sin embargo, en este caso el resultado de la consulta no se materializa en la base de datos, sencillamente se muestra. El proceso es el siguiente:

- Se calculan las relaciones necesarias usando el grafo de dependencias.
- Se extienden las definiciones de las relaciones necesarias y se obtienen las tuplas para ellas.
- La consulta es un nuevo `sel_stm` que es, como en el caso de las vistas, el resultado de remplazar `R` por `SESSION.R` para las relaciones necesarias dentro de la consulta.
- Mediante un cursor de SQL PL se lanza la consulta `sel_stm` a la base de datos, se obtiene la respuesta y se muestra.
- Las tablas temporales son descartadas y el resultado de la consulta no se materializa.

El lado derecho de la definición de la vista `reachable` es un ejemplo válido de consulta en el sistema *HR-SQL*. El proceso es el mismo que hemos presentado en la sección anterior con la salvedad de que, en vez de almacenar la vista `reachable` en la base de datos, se muestran las tuplas por pantalla como resultado de emitir la consulta `sel_stm` (con sus correspondientes reemplazamientos) al SGBDR.

Cuando se formula una consulta no hipotética del lenguaje SQL estándar, el sistema la lanza directamente al SGBDR subyacente sin pasar por ningún otro procesamiento, como se puede ver en la figura 3.9 al comienzo de esta sección.

A continuación, terminamos el capítulo presentando algunos resultados de eficiencia.

3.6. Análisis de rendimiento

En esta sección (extraída de [B.3]) se presentan resultados de rendimiento del sistema. En primer lugar mostramos la mejora de eficiencia del sistema por el uso del algoritmo *in/out* explicado en la sección 3.5.2. En segundo lugar hacemos una comparativa entre distintos SGBDR actuales introduciendo las ventajas de una optimización *semi-naïve* para consultas recursivas lineales basada en [121].

En esta sección utilizamos milisegundos como medida del tiempo de ejecución. Para medir el rendimiento se usa la media del número de ejecuciones de un programa eliminado el máximo y el mínimo de los tiempos tomados.

Análisis del algoritmo *in/out*

Para medir el rendimiento tomamos como caso de prueba la relación `reach` que implementa el cierre transitivo de la relación `flight` presentada en la sección 3.5 (usamos el termino *benchmark* para referirnos a este caso de pruebas siguiendo la nomenclatura inglesa). Se recuerda a continuación la definición de `reach`:

```
reach(frm integer, to integer) :=
    select flight.frm, flight.to from flight union
    select reach.frm,  flight.to from reach,flight
      where reach.to = flight.frm;
```

Como mostramos, hemos cambiado el tipo de los campos de `varchar` al tipo numérico `integer`. Es decir, en adelante consideramos las conexiones entre los vuelos como las tuplas $\{(1,2),(2,3),\ldots,(n,n+1)\}$ donde $n+1$ es el número de nodos en el grafo.

El cuadro 3.1 muestra los resultados de este *benchmark* con un número de tuplas de entrada (en la relación `flight`) que varía desde 100 hasta 350 en la primera columna. La segunda columna presenta el número de tuplas generadas en el resultado. La tercera y la cuarta columna muestran respectivamente el tiempo necesario para resolver la consulta en *HR-SQL* sin la mejora del algoritmo *in/out* (lo llamaremos sin FOI utilizando este acrónimo para representar sus siglas en inglés: *Factoring-Out Improvement*) y con esta mejora (con FOI) para el caso contrario. La quinta columna (*speed-up*) presenta la ganancia de velocidad debido a FOI.

Estos *benchmarks* han sido probados en un ordenador con CPU Intel Core2 Quad a 2.4GHz y 3GB RAM. Se ha usado el sistema operativo Windows XP 32bits SP3 y el servidor de base de datos IBM DB2 Express Edition 10.1.0 con la configuración estándar por defecto.

Tuplas	tuplas resultantes	sin FOI	con FOI	*speed-up*	diferencia
100	5.050	1.135	1.050	8.1 %	85
150	11.325	4.438	3.428	29,4 %	1.010
200	20.100	10.048	8.172	23,0 %	1.876
250	31.375	19.001	16.041	18,5 %	2.960
300	45.150	32.710	28.381	15,3 %	4.329
350	61.425	50.085	44.175	13,4 %	5.910

Cuadro 3.1: Resultados de la mejora FOI.

Como resultado de estas pruebas se confirma la mejora de rendimiento debida a extraer `select * from flight` en la definición de `reach` (dentro del bucle **repeat**). Alcanzamos una mejora de hasta casi un 30 % solamente extrayendo este fragmento. Sin embargo, a medida que el número de tuplas se va incrementando, la mejora disminuye dado que se diluye en comparación con el tiempo requerido para ejecutar el bucle **repeat** (debido a la ejecución del operador **except** dentro de él).

En la siguiente sección hacemos una comparativa con otros sistemas deductivos y relacionales.

Análisis del sistema

En esta sección incluimos en la comparativa, además de *HR-SQL*, otros sistemas de bases de datos actuales que incluyen consultas recursivas: PostgreSQL 9.3, Oracle 11g, y DB2 10.1. Trabajamos con todos ellos con su configuración por defecto.

Se utiliza también el *benchmark* de la sección anterior. Además presentamos una optimización en el cálculo de la recursión (siguiendo la aproximación de la optimización diferencial *semi-naïve* de [121]) orientada a mejorar los resultados de rendimiento.

Para hacer una comparación justa entre HR-SQL y otros SGBDR que no descartan duplicados omitimos el operador **except**.

Además incluimos en la comparativa los tiempos de respuesta de la última versión del sistema DLV^{DB}. Se trata de un sistema deductivo capaz también de trabajar con diferentes SGBDR (utilizando también la conexión ODBC) y calcular el cierre transitivo. Sin embargo, este sistema calcula el resultado a partir de un programa lógico, en lugar de usar SQL.

Los valores obtenidos aparecen en el cuadro 3.2 en el eje horizontal. Las filas incluyen los SGBDR que consideramos (primera columna), el sistema concreto conectado con el SGBDR de la columna anterior (segunda columna), y en las cinco columnas restantes los tiempos de respuesta para cada instancia (desde 100 hasta 500 tuplas en la relación `flight`, que devuelven desde 5.050 hasta 125.250 como respuesta a la consulta que utilizamos como *benchmark*).

SGBDR	Sistema	100	200	300	400	500
PostgreSQL	*SQL nativo*	161	187	240	360	713
	HR-SQL	500	3.198	12.406	39.802	71.922
	Diff-HR-SQL	208	459	1.073	2.271	4.115
	TDiff-HR-SQL	260	578	1.323	2.745	5.693
	DLV^{DB}	703	1.651	4.458	8.047	13.120
Oracle	*SQL nativo*	604	1.781	5.765	13.349	26.297
	HR-SQL	880	3.802	12,057	27.989	56.641
	Diff-HR-SQL	708	1.437	3.224	6.240	11.469
	TDiff-HR-SQL	646	995	1.708	2.453	3.422
	DLV^{DB}	6.875	12.849	18.912	30.583	42.146
DB2	*SQL nativo*	677	1.016	1.323	2.052	3.099
	HR-SQL	1.271	5.797	97.052	129.917	150.104
	Diff-HR-SQL	698	932	2.672	2,859	3.213
	TDiff-HR-SQL	646	1.000	1.578	4.021	9.021
	DLV^{DB}	6.339	12.666	53.552	57.349	100.391

Cuadro 3.2: Análisis de los sistemas

A cada fila SGBDR (*PostgreSQL*, *Oracle*, *DB2*)[2] le corresponden cinco filas que representan los sistemas concretos. Dentro de estos, la primera fila SQL nativo representa al SGBDR corresponde a la ejecución nativa del *benchmark*, i.e., trabajando con la formulación del cierre transitivo que cada SGBDR admite.

Por ejemplo, DB2 utiliza la siguiente sintaxis para el *benchmark* propuesto (donde `rec` es la relación recursiva temporal que utilizamos para construir la relación `reach`):

```
INSERT INTO reach
  WITH rec(frm,to) AS
    (SELECT * FROM flight
     UNION ALL
     SELECT flight.frm, rec.to FROM flight,rec
     WHERE flight.to = rec.frm)
   SELECT * FROM rec;
```

[2]MySQL no soporta ningún tipo de consultas recursivas.

La siguiente fila presenta los resultados del sistema *HR-SQL*. En la fila *Diff-HR-SQL* aparecen los resultados para *HR-SQL* con la optimización diferencial *semi-naïve*. *Grosso modo* podemos decir que, en el contexto de una consulta recursiva lineal, esta optimización utiliza para generar tuplas resultantes de cada iteración solo las generadas en la iteración anterior [121].

Para obtener este comportamiento hemos añadido un parámetro `IT` en el *benchmark* de *HR-SQL* donde se guarda la iteración en que cada tupla ha sido generada:

```
INSERT INTO reach
  SELECT flight.frm, reach.to, IT
  FROM flight, reach
  WHERE flight.to = reach.frm AND
        reach.it = IT-1;
```

A continuación, la fila *TDiff-HR-SQL* representa una implementación alternativa del algoritmo de optimización diferencial *semi-naïve* que consiste en almacenar las tuplas generadas en cada iteración en una tabla temporal. Después el resultado de cada iteración se calcula haciendo la reunión (*join*) entre la tabla `flight` y dicha tabla temporal. Así se evita hacer, cada vez dentro del bucle, el recuento del número de tuplas para la relación `reach`, que crece sucesivamente en cada iteración. Por tanto utilizamos dos tablas temporales: una para acceder a las tuplas que se generan en la iteración anterior y otra para almacenar las nuevas tuplas. En el ejemplo siguiente `reach_temp1` almacena las tuplas generadas en la iteración anterior y `reach_temp2` se usa para la iteración actual.

A continuación mostramos las instrucciones SQL (incluidas en un *script*) que se envían a DB2 en cada iteración (de nuevo, las tablas temporales se etiquetan con la palabra reservada `SESSION`):

```
INSERT INTO SESSION.reach_temp2
  SELECT flight.ori, SESSION.reach_temp1.des
  FROM flight, SESSION.reach_temp1
  WHERE flight.des = SESSION.reach_temp1.ori;
...
INSERT INTO reach SELECT * FROM SESSION.reach_temp1;
DELETE FROM SESSION.reach_temp1;
INSERT INTO SESSION.reach_temp1
  SELECT * FROM SESSION.reach_temp2;
DELETE FROM SESSION.reach_temp2;
```

La primera instrucción SQL guarda en `reach_temp2` el resultado que acaba de ser calculado para la iteración actual. Las siguientes instrucciones cargan en la tabla `reach` el resultado de la iteración anterior y trasladan el resultado calculado en `reach_temp2` a `reach_temp1` de forma que estarán disponibles para la siguiente iteración. Por otro lado se borra `reach_temp2` para prepararlo también para la próxima iteración.

El uso de tablas temporales supone una mejora de rendimiento dado que no demanda generar entradas de *log* ni gestión de concurrencia. Se calculan en memoria principal hasta agotarla y, en caso de que no haya suficiente memoria RAM, se utiliza la memoria secundaria.

Atendiendo a los números obtenidos, podemos destacar que los mejores resultados de rendimiento los obtiene SQL nativo en PostgreSQL para todas las instancias consideradas del *benchmark*. También que los peores resultados corresponden a *HR-SQL* sin optimización

(incluyendo el operador **except**), lo que también se debe a que la reunión y la diferencia deben procesarse en cada iteración para todas las tuplas, incluyendo las que no se usarán para generar otras nuevas. La optimización diferencial *semi-naïve* (que evita también el operador **except**) soluciona este problema en gran medida, con un factor de 150.104/3.213=46,7×, al comparar *HR-SQL* con *Diff-HR-SQL* para DB2. DLV^{DB} es el siguiente sistema más eficiente. Se comporta mejor que *HR-SQL* pero peor que el resto. Dependiendo del SGBDR concreto, el siguiente mejor resultado lo obtiene o bien *Diff-HR-SQL* o *TDiff-HR-SQL*: el primero obtiene mejores resultados que el segundo para PostgreSQL y ocurre lo contrario para Oracle y DB2. Ambos obtienen mejores resultados que *SQL nativo* para Oracle y *Diff-HR-SQL* se comporta de forma similar a DB2.

Estos números demuestran cómo se comportan técnicas similares gestionadas de forma diferente por cada SGBDR concreto. Además el uso de tablas temporales es de suma importancia para ahorrar tiempo en Oracle y tiene el efecto contrario, a la vista de los resultados, en DB2.

Teniendo todo esto en cuenta, en el mejor caso podemos competir con un SGBDR con un factor de 26.297/3.422=7,7× y considerando el peor caso (con la mejor optimización) este resultado sería de un factor de 4.115/713=5,8×.

Para entender mejor esta diferencia debemos tener en cuenta que el sistema *HR-SQL* ejecuta un script interpretado (Python) y en cada iteración varias sentencias SQL se envían al SGBDR utilizando la conexión ODBC lo que supone una penalización considerable del rendimiento.

Finalmente, podemos concluir que el mecanismo de cálculo de las bases de datos *HR-SQL* es un buen punto de partida para explorar nuevas mejoras de rendimiento.

Con la presentación de *HR-SQL* concluimos el tercer capítulo de la tesis que resume los contenidos que aparecen en las publicaciones asociadas [B.1,B.2,B.3]. A continuación, en el último capítulo de la memoria, presentamos la conclusiones y planteamos el trabajo futuro.

Publicaciones asociadas al capítulo 3

[B.1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Formalizing a Broader Recursion Coverage in SQL.
En *Symposium on Practical Aspects of Declarative Languages (PADL'13)*, volumen 7752 de *LNCS*, páginas 93 – 108, 2013.
→ **Página** 176

[B.2] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Incorporating Hypothetical Views and Extended Recursion into SQL Database Systems.
En Ken Mcmillan, Aart Middeldorp, Geoff Sutcliffe, y Andrei Voronkov, editores, *LPAR-19*, volumen 26 de *EPiC Series*, páginas 9–22. EasyChair, 2014.
→ **Página** 192

[B.3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
R-SQL: An SQL Database System with Extended Recursion.
En *Electronic Communications of the EASST*, volumen 64: Programming and Computer Languages, páginas 1–18, 2013.
→ **Página** 206

Capítulo 4

Conclusiones y trabajo futuro

En la actualidad encontramos multitud de aplicaciones de bases de datos para la gestión de todo tipo de empresas, entornos científicos e instituciones públicas. Son una parte esencial de cualquier negocio o actividad dado que en ellas se almacenan los datos estratégicos para su funcionamiento. El modelo relacional es el más extendido y utilizado hoy en día para implementar bases de datos. Sin embargo, el estudio de lenguajes de BDD como Datalog [103, 101, 41] ha cobrado importancia en los últimos años por su sencillez y potencia expresiva.

En esta tesis hemos contribuido en estas dos áreas: por un lado, en el área de las BDD con restricciones, hemos presentando, implementado y fundamentando el lenguaje $HH_{\neg}(\mathcal{C})$ que aporta, además de un lenguaje de restricciones más rico que los habituales, nuevas capacidades que no permite el lenguaje de referencia Datalog con restricciones [95, 64]. En concreto hemos incorporado cuantificadores universales y consultas hipotéticas que pueden aparecer incluso en relaciones recursivas.

Por otro lado, dentro de las BDR, hemos propuesto el lenguaje *HR-SQL* como extensión del lenguaje estándar SQL [36], sus fundamentos semánticos y su implementación. Nuestro lenguaje maneja razonamiento hipotético e incorpora un tratamiento más general de la recursión que permite recursión mutua y recursión no lineal.

Con respecto a $HH_{\neg}(\mathcal{C})$ hemos presentado:

- Una formalización del lenguaje que incluye, ente otras facilidades, implicación intuicionista [72] para expresar consultas hipotéticas, cuantificadores universales y existenciales explícitos. También se ha presentado una semántica operacional de punto fijo por estratos para el lenguaje que sirve de guía para la implementación de un sistema concreto. Además hemos demostrado la corrección y completitud entre la semántica operacional y la semántica de pruebas definida previamente en [83]. El hecho de incorporar las hipótesis en consultas y predicados de la base de datos complica el cómputo del punto fijo frente al de otros sistemas de BDD. Sin embargo, la extensión de la noción de grafo de dependencias en el que se basa una estratificación para los predicados se ha demostrado eficaz para llevar a cabo el cómputo de forma correcta. La combinación de la semántica estratificada con el uso de un sistema de restricciones compacto garantiza la terminación del cómputo.

- Un sistema de bases de datos que implementa la semántica de punto fijo del lenguaje $HH_{\neg}(\mathcal{C})$. En el sistema destacamos la primera implementación que incorpora consultas hipotéticas basadas en la implicación intuicionista para un lenguaje de BDD. Además el marco teórico ha resultado adecuado para incluir en el sistema funcionalidades habituales

de los lenguajes relacionales:

- Hemos incorporado las funciones de agregación al sistema. Con este fin se ha propuesto usar un enfoque similar al utilizado para incorporar la negación en las bases de datos deductivas basado en la estratificación (capítulo 3 de [122]). Hemos implementado la función de recuento sobre un predicado y también funciones de sumatorio, media, mínimo y máximo sobre las variables de un predicado. El uso de la estratificación permite definir consultas hipotéticas con funciones de agregación asegurando la corrección de los datos incluso en los cómputos locales.
- Hemos implementado restricciones de integridad fuertes que garantizan un uso consistente de la base de datos. En concreto hemos implementado las restricciones de clave primaria, clave ajena y dependencias funcionales. El hecho de disponer del sistema de restricciones en el esquema hace posible que las restricciones de integridad se puedan expresar de forma sencilla y que se puedan trasladar de forma directa a la implementación.

Con respecto a *HR-SQL* hemos presentado:

- Una semántica de punto fijo por estratos inspirada en la semántica de $HH_{\neg}(\mathcal{C})$. Proponemos trasladar ciertas técnicas declarativas para dar semántica a un lenguaje de BDR. Cuando se formula una consulta hipotética se utiliza también la noción de grafo de dependencias para determinar las relaciones de la base de datos original que se ven afectadas y deben ser recalculadas. Dado que las hipótesis están limitadas a la definición de vistas y consultas, se aprovecha parte del cálculo del punto fijo (y se extiende el grafo calculado) optimizando el cómputo de estas. De esta forma presentamos uno de los pocos formalismos teóricos para un lenguaje de BDR y además lo extendemos con capacidades de las BDD.
- El sistema *HR-SQL* traslada características que provienen del marco formal a un SGBDR concreto. Hemos propuesto una extensión del lenguaje SQL para incorporar razonamiento hipotético y un tratamiento más general de la recursión que incluye recursión no lineal, recursión mutua y evita cómputos no terminantes al calcular el significado de relaciones que definen grafos con ciclos. En la actualidad muchos sistemas gestores aceptan la limitada expresión de recursión del estándar SQL (como PostgreSQL, DB2 u Oracle) y otros directamente no admiten ningún tipo de definiciones recursivas en sus relaciones (como MySQL o Access). Tampoco el razonamiento hipotético aparece en ningún sistema de BDR de ámbito comercial. *HR-SQL* permite recursión extendida e hipótesis en vistas y consultas que se puede incorporar a la mayoría de los SGBDR. El único requisito para extender un SGBDR con *HR-SQL* es que permita acceso mediante Python o un lenguaje de cuarta generación. Además hemos presentado una comparativa del rendimiento del sistema con respecto a otros sistemas de BDD y de otros de BDR. Finalmente, *HR-SQL* utiliza el algoritmo *in/out* que reduce las llamadas al sistema gestor limitando la generación e inserción de tuplas dentro de los bucles que calculan el punto fijo a la parte recursiva de las relaciones (en caso de que exista).

Como se ha visto, el uso de técnicas de bases de datos deductivas y de transformación de programas para la mejora del rendimiento es un campo de estudio abierto del que se obtienen buenos resultados. Los estudios que hemos llevado a cabo y los marcos teóricos que hemos propuesto permiten incorporar nuevas funcionalidades a cualquier SGBDR que además se presentan respaldadas por un fundamento consistente.

A partir de las contribuciones conseguidas planteamos algunas líneas de trabajo futuro a corto plazo. Del estudio desarrollado para extender sistemas gestores de BDR con un sistema implementado en SWI-Prolog [129] se puede mejorar la eficiencia de $HH_{\neg}(\mathcal{C})$ si delegamos determinadas partes del cómputo del punto fijo en el SGBDR. En concreto aprovechando la eficiencia de los sistemas relacionales actuales se podría delegar el cómputo de la parte extensional, de la parte no recursiva y de la parte no hipotética de las cláusulas de $HH_{\neg}(\mathcal{C})$ en un sistema relacional para los casos que presentan peor rendimiento.

Para *HR-SQL* proponemos, en primer lugar, integrar en el sistema la optimización diferencial *semi-naïve* siguiendo la aproximación de [121] que mejora el rendimiento del sistema como hacemos con el *benchmark* de la sección 3.6. Siguiendo esta línea, existen además métodos de optimización de la recursión lineal [84] que se pueden aplicar con facilidad a nuestro sistema haciendo uso también del grafo de dependencias junto con un estudio del tipo de consulta (e.g., si conlleva un cierre transitivo, si contiene agregados o si genera duplicados, etc.). La combinación de estos dos métodos sería un buen objeto de estudio para analizar si redunda en una mejora de rendimiento.

Finalmente, otro avance a corto plazo que planteamos es permitir vistas hipotéticas en *HR-SQL* sin necesidad de materializarlas. En concreto se podría integrar el proceso de cálculo de punto fijo en el propio SGBDR sin necesidad de un *front-end* como se ha hecho hasta ahora. Para ello proponemos utilizar las funciones de tablas disponibles en algunos sistemas gestores (como por ejemplo en DB2). El objetivo de esta propuesta es obtener resultados *on-the-fly* como hacen los SGBDR para gestionar las vistas. Además el uso de las tablas temporales parece también adecuado para obtener buenos resultados de eficiencia como hemos visto al usarlas para procesar vistas y consultas hipotéticas.

Otra línea de investigación a medio plazo que planteamos es la mejora de eficiencia de *HR-SQL* utilizando otras técnicas diferentes a las que hemos usado como son el uso de las transformaciones *magic sets* [10] o técnicas de *tabling* [116] que podrían ser adaptadas para nuestra implementación, al igual que hacen otros sistemas actuales [103, 104, 19].

Finalmente, como trabajo a largo plazo, proponemos trasladar más características expresivas de $HH_{\neg}(\mathcal{C})$ a *HR-SQL* como son las restricciones. La representación de las restricciones como filas de las tablas relacionales puede aportar aún mayor capacidad expresiva.

Bibliografía

[1] Almendros-Jiménez, J.M. y A. Becerra-Terón: *A Relational Algebra for Functional Logic Deductive Databases*. En *Proc. of 5th International Conference on Perspectives of System Informatics, PSI03*, número 2890 en *LNCS*, páginas 494–508. Springer, 2003.

[2] Apt, K. R., H. A. Blair y A. Walker: *Towards a Theory of Declarative Knowledge*. Foundations of deductive databases and logic programming, páginas 89–148, 1988.

[3] Arni, F., K. Ong, S. Tsur, H. Wang y C. Zaniolo: *The Deductive Database System LDL++*. TPLP, 3(1):61–94, 2003.

[4] Arruabarrena, R., P. Lucio y M. Navarro: *A Strong Logic Programming View for Static Embedded Implications*. En *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, páginas 56–72, 1999.

[5] Balbin, I., G. S. Port, K. Ramamohanarao y K. Meenakshi: *Efficient Bottom-UP Computation of Queries on Stratified Databases*. J. Log. Program., 11(3&4):295–344, 1991.

[6] Balbin, I. y K. Ramamohanarao: *A Generalization of the Differential Approach to Recursive Query Evaluation*. J. Log. Program., 4(3):259–262, 1987.

[7] Balmin, A., T. Papadimitriou y Y. Papakonstantinou: *Hypothetical Queries in an OLAP Environment*. En *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, páginas 220–231, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc., ISBN 1-55860-715-3. `http://dl.acm.org/citation.cfm?id=645926.672016`.

[8] Bancilhon, F., D. Maier, Y. Sagiv y J. D Ullman: *Magic sets and other strange ways to implement logic programs*. En *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, páginas 1–15, New York, NY, USA, 1986. ACM, ISBN 0-89791-179-2.

[9] Baudinet, M., M. Nizette y P. Wolper: *On the Representation of Infinite Temporal Data and Queries*, 1991.

[10] Beeri, C. y R. Ramakrishnan: *On the power of magic*. Journal of Logic Programming, 10(3-4):255–299, 1991, ISSN 0743-1066.

[11] Bertino, E., B. Catania y R. Gori: *Enhancing the Expressive Power of the U-Datalog Language*. Theory and Practice of Logic Programming, 1(1):105–122, 2001.

[12] Bocca, J. B.: *MegaLog - A platform for developing Knowledge Base Management Systems*. En Makinouchi, Akifumi (editor): *Database Systems for Advanced Applications '91, Proceedings of the Second International Symposium on Database Systems for Advanced Applications, Tokyo, Japan, April 2-4, 1991*, volumen 2 de *Advanced Database Research and Development Series*, páginas 374–380. World Scientific, 1991.

[13] Bonner, A. J.: *Hypothetical Datalog: Complexity and Expressibility*. Theoretical Computer Science, 76:144–160, 1988.

[14] Bonner, A. J.: *A Logical Semantics For Hypothetical Rulebases With Deletion*, 1997.

[15] Bonner, A. J. y L. T. McCarty: *Adding Negation-as-Failure to Intuitionistic Logic Programming*. En Lusk, Ewing L. y Ross A. Overbeek (editores): *Logic Programming, Proc. of the North American Conference*, páginas 681–703. The MIT Press, 1989. `citeseer.ist.psu.edu/bonner92adding.html`.

[16] Bonner, A. J., L. T. McCarty y K. Vadaparty: *Expressing Database Queries with Intuitionistic Logic*. En Lusk, Ewing L. y Ross A. Overbeek (editores): *Proceedings of the North American Conference on Logic Programming*, páginas 831–850, 1989, ISBN 0-262-62064-2. `citeseer.ist.psu.edu/bonner89expressing.html`.

[17] Bry, F., H. Decker y R. Manthey: *A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases*. En *EDBT '88: Proceedings of the International Conference on Extending Database Technology*, páginas 488–505, London, UK, 1988. Springer-Verlag, ISBN 3-540-19074-0.

[18] Byon, J.H. y P. Z. Revesz: *DISCO: A Constraint Database System with Sets*. En *In CONTESSA Workshop on Constraint Databases and Applications*, páginas 68–83. Springer-Verlag, 1995.

[19] Cabeza, D. y M. Hermenegildo: *The Ciao Module System: A New Module System for Prolog* . ENTCS, 30(3):122 – 142, 2000, ISSN 1571-0661. Parallelism and Implementation Technology for (Constraint) Logic Programming (in connection with ICLP'99, International Conference on Logic Programming).

[20] Calì, A., G. Gottlob y T. Lukasiewicz: *Datalog±: a unified approach to ontologies and integrity constraints*. En *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, páginas 14–30, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-423-2.

[21] Chandra, A. K. y D. Harel: *Horn Clauses Queries and Generalizations*. J. Log. Program., 2(1):1–15, 1985.

[22] Chang, C.L.: *DEDUCE 2: Further Investigation of Deduction in Relational Data Bases*. En *IBM, Res.R. No.RJ2147, San Jose; ACM Computing Reviews 40,416.* ACM Computing Reviews, Mayo 1978.

[23] Chimenti, D., R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur y C. Zaniolo: *The LDL System Prototype*. IEEE Transactions on Knowledge and Data Engineering, 2:76–90, 1990.

[24] Christiansen, H. y T. Andreasen: *A Practical Approach to Hypothetical Database Queries*. En *Transactions and Change in Logic Databases*, volumen 1472 de *LNCS*, páginas 340–355. Springer, 1998, ISBN 3-540-65305-8.

[25] Codd, E. F.: *Data Base Sublanguage Founded on the Relational Calculus*. IBM Research Report, San Jose, California, RJ893, 1971.

[26] Codd, E. F.: *A Database Sublanguage Founded on the Relational Calculus*. En *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971*, páginas 35–68, 1971.

[27] Codd, E. F.: *Relational Completeness of Data Base Sublanguages*. In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California, 1972. db/labs/ibm/RJ987.html.

[28] Codd, E.F.: *A Relational Model for Large Shared Databanks*. Communications of the ACM, 13(6):377–390, June 1970.

[29] Correas, J., J. M. Gómez, M. Carro, D. Cabeza y M. V. Hermenegildo: *A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations)*. En Jayaraman, Bharat (editor): *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volumen 3057 de *Lecture Notes in Computer Science*, páginas 104–119. Springer, 2004, ISBN 3-540-22253-7. http://dx.doi.org/10.1007/978-3-540-24836-1_8.

[30] Date, C. J.: *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.

[31] Dell'Armi, T., W. Faber, G. Ielpa, N. Leone y G. Pfeifer: *Aggregate Functions in DLV*. En *Answer Set Programming 2003 (SP03)*, Messina, Sicily, september, 26-28 2003.

[32] Emden, M. H. Van y R. A. Kowalski: *The Semantics of Predicate Logic as a Programming Language*. J. ACM, 23(4):733–742, 1976.

[33] Finkelstein, S. J., N. Mattos, I. S. Mumick y H. Pirahesh: *Expressing Recursive Queries in SQL*. Informe técnico, ISO, 1996.

[34] García-Díaz, M. y S. Nieva: *Solving Constraints for an Instance of an Extended CLP Language over a Domain based on Real Numbers and Herbrand Terms*. Journal of Functional and Logic Programming, 2003(2), September 2003.

[35] García-Díaz, M. y S. Nieva: *Providing Declarative Semantics for HH Extended Constraint Logic Programs*. En *Proceedings of the 6th ACM SIGPLAN Int. Conf. on PPDP*, páginas 55 – 66, 2004.

[36] Garcia-Molina, H., J. D. Ullman y J. Widom: *Database systems - the complete book (2. ed.)*. Pearson Education, 2009, ISBN 978-0-13-187325-4.

[37] Gelfond, M. y V. Lifschitz: *The Stable Model Semantics For Logic Programming*. En *ICLP/SLP*, páginas 1070–1080. MIT Press, 1988.

[38] Golfarelli, M. y S. Rizzi: *What-if Simulation Modeling in Business Intelligence*. IJDWM, 5(4):24–43, 2009.

[39] Greco, S.: *Dynamic Programming in Datalog with Aggregates*. IEEE Trans. on Knowl. and Data Eng., 11(2):265–283, 1999, ISSN 1041-4347.

[40] Green, C. C. y B. Raphael: *The use of theorem-proving techniques in question-answering systems*. En *ACM '68: Proceedings of the 1968 23rd ACM national conference*, páginas 169–181, New York, NY, USA, 1968. ACM.

[41] Green, T. J.: *LogiQL: A Declarative Language for Enterprise Applications*. En *Proceedings of the 34th ACM Symposium on Principles of Database Systems*, PODS '15, páginas 59–64, New York, USA, 2015. ACM, ISBN 978-1-4503-2757-2. `http://doi.acm.org/10.1145/2745754.2745780`.

[42] Griffin, T. y R. Hull: *A Framework for Implementing Hypothetical Queries*. En *SIGMOD Conference*, páginas 231–242, 1997.

[43] Grosky, W. I. y R. Mehrotra: *Introduction: Image Database Management*. Computer, 22(12):7–8, 1989, ISSN 0018-9162.

[44] Grumbach, S., P. Rigaux, L. Chesnay y L. Segoufin: *Spatio-Temporal Data Handling with Constraints*, 1998.

[45] Grumbach, S., P. Rigaux, L. Chesnay y L. Segoufin: *The DEDALE System for Complex Spatial Queries*, 1998.

[46] Günther, O. y J. Bilmes: *Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation*. IEEE Trans. on Knowl. and Data Eng., 3(3):342–356, 1991, ISSN 1041-4347.

[47] Hansen, M. R., B. S. Hansen, P. Lucas y P. van Emde Boas: *Integrating relational databases and constraint languages*. Comput. Lang., 14(2):63–82, 1989, ISSN 0096-0551.

[48] Hargrove, W. W., R. H. Gardner, M. G. Turner, W. H. Romme y D. G. Despain: *Simulating fire patterns in heterogeneous landscapes*, 2000.

[49] Henschen, L. J. y S. A. Naqvi: *On compiling queries in recursive first-order databases*. J. ACM, 31(1):47–85, 1984, ISSN 0004-5411.

[50] Inmon, W. H.: *Building the data warehouse*. QED Information Sciences, Inc., Wellesley, MA, USA, 2005.

[51] ISO/IEC: *SQL:1986 ISO/IEC 1986 Standard*, 1986.

[52] Jaffar, J. y J. L. Lassez: *Constraint Logic Programming*. En *14th ACM Symp. on Principles of Programming Languages (POPL'87)*, páginas 111–119, Munich, Germany, January 1987. ACM Press.

[53] Jagadish, H. V.: *A retrieval technique for similar shapes*. En *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, páginas 208–217, New York, NY, USA, 1991. ACM, ISBN 0-89791-425-2.

[54] Jarke, M., M. A. Jeusfeld y C. Quix: *ConceptBase V7.1 User Manual*. Informe técnico, RWTH Aachen, April 2008.

[55] Jeusfeld, M. y M. Jarke: *From Relational to Object-Oriented Integrity Simplification*, 1991.

[56] Jeusfeld, M. y M. Staudt: *Query Optimization in Deductive Object Bases*, 1993.

[57] Kabanza, F., J m. Stevenne y P. Wolper: *Handling Infinite Temporal Data*. En *Journal of Computer and System Sciences*, páginas 392–403, 1990.

[58] Kanellakis, P. C., G. M. Kuper y P. Z. Revesz: *Constraint Query Languages*. En *Symposium on Principles of Database Systems*, páginas 299–313, 1990.

[59] Kanjamala, P., P. Z. Revesz y Y. Wang: *MLPQ/GIS: A GIS using Linear Constraint Databases*. En *Proc. Ninth International Conference on Management of Data*, páginas 389–393. McGraw Hill, 1998.

[60] Kellogg, C. y L. Travis: *Reasoning with Data in a Deductively Augmented Data Management System*. En *Advances in Data Base Theory*, páginas 261–295, 1979.

[61] Kemp, D. B., K. Ramamohanarao, I. Balbin y K. Meenakshi: *Propagating Constraints in Recusive Deduction Databases*. En *NACLP*, páginas 981–998, 1989.

[62] Kowalski, R. A.: *Logic for Data Description*. En *Logic and Data Bases*, páginas 77–103, 1977.

[63] Kowalski, R. A., F. Sadri y P. Soper: *Integrity Checking in Deductive Databases*. En *In Proceedings of the VLDB International Conference*, páginas 61–69. Morgan Kaufmann Publishers, 1987.

[64] Kuper, G., L. Libkin y J. Paredaens (editores): *Constraint Databases*. Springer, 2000.

[65] Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin y C. Unkel: *Context-sensitive program analysis as database queries*. En Li, Chen (editor): *PODS*, páginas 1–12. ACM, 2005, ISBN 1-59593-062-0.

[66] Leach, J., S. Nieva y M. Rodríguez-Artalejo: *Constraint Logic Programming with Hereditary Harrop Formulas*. TPLP, 1(4):409–445, 2001.

[67] Lefebvre, A.: *Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases*. New Generation Comput., 12(2):131–160, 1994.

[68] Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri y F. Scarcello: *The DLV system for knowledge representation and reasoning*. ACM Trans. Comput. Log., 7(3):499–562, 2006.

[69] Lifschitz, V.: *Introduction to Answer Set Programming*. Introductory course at the 16th European Summer School in Logic, Language and Information. Unpublished Draft, 2004. Available at www.cs.utexas.edu/users/vl/mypapers/esslli.ps, 2004.

[70] Maher, M. J. y R. Ramakrishnan: *D'eja vu in fixpoints of logic programs*. En *in Proceedings of the North American Conference on Logic Programming*, páginas 963–980. The MIT Press, 1989.

[71] Maluszynski, J. y A. Szalas: *Logical Foundations and Complexity of 4QL, a Query Language with Unrestricted Negation*. CoRR, abs/1011.5105, 2010. `http://arxiv.org/abs/1011.5105`.

[72] McCarty, L. Thorne: *Clausal Intuitionistic Logic I - Fixed-Point Semantics*. J. Log. Program., 5(1):1–31, 1988.

[73] Melton, J. y A. R. Simon: *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, Mayo 2001, ISBN 1558604561.

[74] Miller, D.: *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*. J. Log. Comput., 1(4):497–536, 1991. `http://dx.doi.org/10.1093/logcom/1.4.497`.

[75] Miller, D., G. Nadathur, F. Pfenning y A. Scedrov: *Uniform Proofs as a Foundation for Logic Programming*. Annals of Pure and Applied Logic, 51:125–157, 1991.

[76] Minker, J.: *Perspectives in Deductive Databases*. En *PODS*, página 135, 1987.

[77] Minker, J.: *Logic and Databases: A 20 Year Retrospective*. En *Logic in Databases*, páginas 3–57, 1996.

[78] Minker, J. y J. M. Nicolas: *On recursive axioms in deductive Databases*. Information Systems, 8(1):1–13, 1983.

[79] Morris, K. A., J. F. Naughton, Y. P. Saraiya, J. D. Ullman y A. Van Gelder: *YAWN! (Yet Another Window on NAIL!)*. IEEE Data Eng. Bull., 10(4):28–43, 1987.

[80] Naqvi, S. y S. Tsur: *A logical language for data and knowledge bases*. Computer Science Press, Inc., New York, NY, USA, 1989, ISBN 0-7167-8200-6.

[81] Nash, J. C.: *The (Dantzig) Simplex Method for Linear Programming*. Computing in Science and Engg., 2(1):29–31, 2000, ISSN 1521-9615.

[82] Naughton, J. F. y R. Ramakrishnan: *How to forget the past without repeating it*. Journal of ACM, 41(6):1151–1177, 1994, ISSN 0004-5411.

[83] Nieva, S., F. Sáenz-Pérez y J. Sánchez: *Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation*. En *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volumen 4989 de *LNCS*, páginas 289–304. Springer Verlag, 2008.

[84] Ordonez, C.: *Optimization of Linear Recursive Queries in SQL*. IEEE Transactions on Knowledge and Data Engineering, 22(2):264–277, 2010, ISSN 1041-4347.

[85] Ramakrishnan, R.: *Magic Templates: A Spellbinding Approach to Logic Programs*. En *Journal of Logic Programming*, páginas 140–159, 1988.

[86] Ramakrishnan, R., D. Srivastava y S. Sudarshan: *Controlling the Search in Bottom-Up Evaluation*. En *In Joint Intl. Conference and Symposium on Logic Programming*, páginas 273–287, 1992.

[87] Ramakrishnan, R., D. Srivastava y S. Sudarshan: *CORAL—Control, Relations and Logic*. En *In Proceedings of the International Conference on Very Large Databases*, páginas 238–250, 1992.

[88] Ramakrishnan, R., D. Srivastava, S. Sudarshan y P. Seshadri: *Implementation of the CORAL Deductive Database System*, 1993.

[89] Ramakrishnan, R. y J.D. Ullman: *A survey of research on Deductive Databases*. The Journal of Logic Programming, 23(2):125–149, 1993.

[90] Ramalingam, G. y E. Visser (editores): *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. ACM, 2007, ISBN 978-1-59593-620-2.

[91] Ramamohanarao, K. y J. Harland: *An introduction to Deductive Database Languages and Systems*. The VLDB Journal, 3(2):107–122, 1994, ISSN 1066-8888.

[92] Reiter, R.: *Towards a Logical Reconstruction of Relational Database Theory*. En *On Conceptual Modelling (Intervale)*, páginas 191–233, 1982.

[93] Reiter, R.: *A Theory of Diagnosis from First Principles*. Artif. Intell., 32(1):57–95, Abril 1987, ISSN 0004-3702. http://dx.doi.org/10.1016/0004-3702(87)90062-2.

[94] Revesz, P. Z.: *Refining Restriction Enzyme Genome Maps*. Constraints, 2(3/4):361–375, 1997, ISSN 1383-7133.

[95] Revesz, P. Z.: *Datalog and Constraints*. En Kuper, G., L. Libkin y J. Paredaens (editores): *Constraint Databases*, capítulo 7, páginas 151–174. Springer, 2000.

[96] Revesz, P. Z.: *Introduction to Constraint Databases*. Springer, 2002.

[97] Revesz, P.Z. y Yiming Li: *MLPQ: a linear constraint database system with aggregate operators*. Database Engineering and Applications Symposium, International, 0:132, 1997.

[98] Robinson, J. A.: *A Machine-Oriented Logic Based on the Resolution Principle*. J. ACM, 12(1):23–41, 1965, ISSN 0004-5411.

[99] Ronen, R. y O. Shmueli: *Evaluating very large datalog queries on social networks*. En *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, páginas 577–587, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-422-5.

[100] Ross, K. A.: *Modular stratification and magic sets for Datalog programs with negation*. J. ACM, 41(6):1216–1266, 1994, ISSN 0004-5411.

[101] Sáenz-Pérez, F.: *Implementing Tabled Hypothetical Datalog*. En *Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'13*, November 2013.

[102] Sáenz-Pérez, F.: *Towards Bridging the Expressiveness Gap Between Relational and Deductive Databases*. En *XIII Jornadas sobre Programación y Lenguajes, PROLE2013 (SISTEDES)*, September 2013.

[103] Sáenz-Pérez, F.: *Datalog Educational System Version 3.10*, January 2015. http://des.sourceforge.net.

[104] Sagonas, K., T. Swift y D. S. Warren: *XSB as an Efficient Deductive Database Engine*. En *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, páginas 442–453. ACM Press, 1994.

[105] Salomon, A.: *Implementation of a Database System with Boolean Algebra Constraints*. Tesis de Doctorado, University of Nebraska, 1998.

[106] Shah, J. J. y M. Mantyla: *Parametric and Feature Based CAD/Cam: Concepts, Techniques, and Applications*. John Wiley & Sons, Inc., New York, NY, USA, 1995, ISBN 0471002143.

[107] Shepherdson, J.C.: *Negation in Logic Programming*. En Minker, J. (editor): *Foundations of Deductive Databases and Logic Programming*, páginas 19–88. Kaufmann, Los Altos, CA, 1988.

[108] Shih, C. y S.W. Dietrich: *Extension Table Evaluation of Datalog Programs with Negation*. En *Proceedings of the IEEE International Phoenix Conference on Computers and Communications*, volumen AZ, páginas 792–798. Scottsdale, March 1991.

[109] Shkapsky, A., M. Yang y C. Zaniolo: *Optimizing recursive queries with monotonic aggregates in DeALS*. En Gehrke, Johannes, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha y Guy M. Lohman (editores): *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, páginas 867–878. IEEE, 2015.

[110] Silberschatz, A., H. Korth y S. Sudarshan: *Database Systems Concepts*. McGraw-Hill, Inc., New York, USA, 5ª edición, 2006, ISBN 0072958863, 9780072958867.

[111] Srivastava, D., R. Ramakrishnan, P. Seshadri y S. Sudarshan: *Coral++: Adding Object-Orientation to a Logic Database Language*. En *In Proceedings of the International Conference on Very Large Data Bases*, páginas 158–170. Morgan Kaufmann Publishers, Inc, 1993.

[112] Stonebraker, M. y K. Keller: *Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System*. En *The 1980 ACM SIGMOD International Conference on Management of Data*, SIGMOD '80, páginas 58–66. ACM, 1980, ISBN 0-89791-018-4. http://doi.acm.org/10.1145/582250.582261.

[113] Sudarshan, S. y R. Ramakrishnan: *Aggregation and Relevance in Deductive Databases*. En *In Proceedings of the International Conference on Very Large Databases*, páginas 501–511, 1991.

[114] Sudarshan, S. y R. Ramakrishnan: *Optimizations of bottom-up evaluation with non-ground terms: extended abstract*. En *ILPS '93: Proceedings of the 1993 international symposium on Logic programming*, páginas 557–574, Cambridge, MA, USA, 1993. MIT Press, ISBN 0-262-63152-0.

[115] Sudarshan, S., D. Srivastava, R. Ramakrishnan y J. F. Naughton: *Space Optimization in the Bottom-Up Evaluation of Logic Programs*. En *in: Proc. SIGMOD*, páginas 5370–6, 1990.

[116] Tamaki, H. y T. Sato: *OLD resolution with tabulation*. En *Proceedings on Third international conference on logic programming*, páginas 84–98, New York, NY, USA, 1986. Springer-Verlag New York, Inc., ISBN 0-387-16492-8.

[117] Tarski, A.: *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[118] Tarski, A.: *A lattice-theoretical fixpoint theorem and its applications*. Pacific Journal of Mathematics, 5:285–309, 1955.

[119] Triska, M.: *Generalising Constraint Solving over Finite Domains*. En *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, páginas 820–821, Berlin, Heidelberg, 2008. Springer-Verlag, ISBN 978-3-540-89981-5. http://dx.doi.org/10.1007/978-3-540-89982-2_89.

[120] Tsur, S. y C. Zaniolo: *LDL: A Logic-Based Data Language*. En Chu, Wesley W., Georges Gardarin, Setsuo Ohsuga y Yahiko Kambayashi (editores): *VLDB'86 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, páginas 33–41. Morgan Kaufmann, 1986, ISBN 0-934613-18-4.

[121] Ullman, J.D.: *Implementation of Logical Query Languages for Databases*. ACM Trans. Database Syst., 10(3):289–321, 1985.

[122] Ullman, J.D.: *Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.

[123] Vaghani, J., K. Ramamohanarao, D. B. Kemp, Z. Somogyi y P.J. Stuckey: *Design Overview of the Aditi Deductive Database System*. En *Proceedings of the Seventh International Conference on Data Engineering*, páginas 240–247, Washington, DC, USA, 1991. IEEE Computer Society, ISBN 0-8186-2138-9.

[124] Van Gelder, A., K. A. Ross y J. S. Schlipf: *The well-founded semantics for general logic programs*. J. ACM, 38(3):619–649, 1991, ISSN 0004-5411.

[125] Vieille, L.: *Recursive Query Processing: Fundamental Algorithms and the DedGin System*. En *Prolog and Databases*, páginas 135–158. World Scientific, 1988.

[126] Wang, H. y C. Zaniolo: *Aggregates in Recursive Datalog and SQL3 Queries*. Informe técnico 980043, IEEE Computer Society, 1998. citeseer.ist.psu.edu/wang98aggregates.html.

[127] Warren, D. S.: *The XWAM: A machine that integrates prolog and deductive databases*. En *Technical Report*, 1989.

[128] Whitney, H.: *Geometric integration theory*. Princeton University Press, Princeton, N. J., 1957.

[129] Wielemaker, J.: *An overview of the SWI-Prolog Programming Environment*. En Mesnard, Fred y Alexander Serebenik (editores): *Proceedings of the 13th International Workshop on Logic Programming Environments*, páginas 1–16, 2003.

[130] Zaniolo, C.: *Key Constraints and Monotonic Aggregates in Deductive Databases*. En *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, páginas 109–134, London, UK, 2002. Springer-Verlag, ISBN 3-540-43960-9.

[131] Zaniolo, C., N. Arni y K. Ong: *Negation and Aggregates in Recursive Rules: the LDL++ Approach*, 1993.

[132] Zhang, Y., H. Chen, H. Sheng y Z. Wu: *Applying Hypothetical Queries to E-Commerce Systems to Support Reservation and Personal Preferences*. En *Proceedings of the 11th International Database Engineering and Applications Symposium*, IDEAS '07, páginas 46–53, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 0-7695-2947-X. http://dx.doi.org/10.1109/IDEAS.2007.15.

[133] Zhou, G., H. Chen y Y. Zhang: *Hypothetical Queries on Multidimensional Dataset*. En Wang, Shouyang, Lean Yu, Fenghua Wen, Shaoyi He, Yong Fang y K. K. Lai (editores): *BIFE*, páginas 539–543. IEEE Computer Society, 2009, ISBN 978-0-7695-3705-4.

Parte II

Publicaciones

Capítulo 5

Publicaciones asociadas al segundo capítulo

[A.1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez.
Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas.
En *Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09)*, páginas 117–128. ACM Press, 2009.
→ **Página** 116

[A.2] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Incorporating Integrity Constraints to a Deductive Database System.
En *XI Jornadas sobre Programación y Lenguajes, PROLE2011 (SISTEDES)*
editores: Purificación Arenas, Victor M. Gulías y Pablo Nogueira, páginas 141–152, Septiembre, 2011.
→ **Página** 128

[A.3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
An Extended Constraint Deductive Database: Theory and implementation.
The Journal of Logic and Algebraic Programming, volumen 21, páginas 20–52, 2013.
→ **Página** 140

Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas

Gabriel Aranda-López
Dpto. de Sistemas Informáticos y Computación,
Univ. Complutense de Madrid
garanda@fdi.ucm.es

Susana Nieva
Dpto. de Sistemas Informáticos y Computación,
Univ. Complutense de Madrid
nieva@sip.ucm.es

Fernando Sáenz-Pérez
Dpto. de Ingeniería del Software e Inteligencia Artificial,
Univ. Complutense de Madrid
fernan@sip.ucm.es

Jaime Sánchez-Hernández
Dpto. de Sistemas Informáticos y Computación,
Univ. Complutense de Madrid
jaime@sip.ucm.es

Abstract

This work is aimed to show a concrete implementation of a deductive database system based on the scheme $HH_{\neg}(\mathcal{C})$ (Hereditary Harrop Formulas with Negation and Constraints) following a fixpoint semantics proposed in a previous work. We have developed a Prolog implementation for this scheme that is constraint system independent, therefore allowing to use it as a base for any instance of the formal scheme. We have developed several specific constraint systems: Real numbers, integers, Boolean and user-defined enumerated types. We have added types to the database so that relations become typed (as tables in relational databases) and each constraint is mapped to its corresponding constraint system. The predicates that compute the fixpoint giving the meaning to a database are described. In particular, we show the implementation of a forcing relation (for derivation steps) and highlight how the inherent difficulties have been overcome in a system allowing hypothetical queries, which make the database dynamically grow.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms Algorithms, Languages, Performance, Theory.

Keywords Hereditary Harrop Formulas, Deductive Databases, Stratification, Constraints, Fixpoint Semantics, Prolog.

1. Introduction

Deductive databases (DDBs) and their query languages have received a great deal of attention recently in many areas, including ontologies [Fikes et al. 2004], the semantic web [Calì et al. 2009], social networks [Ronen and Shmueli 2009], and policy languages [Becker et al. 2007]. The high level expressivity of logic-based query languages has been therefore acknowledged as a useful feature for handling knowledge-based information systems. In particular, Datalog (along its extensions), from which many current references can be found, is playing the role of a renowned language in those settings.

Current deductive database systems (such as, e.g., XSB [Sagonas et al. 1994] –with inputs from the company XSB, Inc.– bddbddb [Lam et al. 2005], LDL++ [Arni et al. 2003], DES [Sáenz-Pérez 2009], ConceptBase [Jarke et al. 2008], .QL [Ramalingam and Visser 2007] –developed by Semmle, Ltd.– and DLV [Leone et al. 2006]) lack several features we provide in the scheme $HH_{\neg}(\mathcal{C})$ (Hereditary Harrop formulas with Negation and Constraints) [Nieva et al. 2008]. These features are helpful for knowledge systems in which more expressive ways of posing queries are needed. The scheme $HH(\mathcal{C})$ was presented in [Leach et al. 2001] as an extension of traditional *LP* (Logic Programming). The one hand, hereditary Harrop formulas extend Horn logic allowing disjunctions, intuitionistic implications and universal quantifiers, improving the expressivity; the other hand, it incorporates the advantages of constraints. Then, $HH_{\neg}(\mathcal{C})$ was obtained by adding negation to the previous scheme in order to conform to the foundations for a DDB, that extends Datalog in the two orthogonal directions, just mentioned.

In our system, a database is a logic program: a set of facts (ground atoms) defining the extensional database, and a set of clauses, defining the intensional database. Clauses can be seen as the definition of views in relational databases. The evaluation of a query with respect to a deductive database can be seen as the computation of a goal from a program (database), and the answer is a constraint. Since the constraint domain is parametric, it is possible to consider different instances (such as arithmetical constraints over real numbers and finite domain constraints).

Let us show the expressivity of our language with the following example written in an instance that allows both real and finite domain constraints.

EXAMPLE 1. Consider the following extensional database for a bank. We follow a syntax similar to Prolog. In addition we write `not` for negation, `=>` for implication, `ex(X,G)` representing $\exists$`X G`, and `fa(X,G)` representing $\forall$`X G`. Some other details of the syntax are deferred to next sections.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

```
% client(Name, Balance, Salary)
client(smith,2000,1200).
client(brown,1000,1500).
client(mcandrew,5300,3000).

% pastDue(Name, Amount)
pastDue(smith,3000).
pastDue(mcandrew,100).

% mortgageQuote(Name, Quote)
mortgageQuote(brown,400).
mortgageQuote(mcandrew,100).
```

where we assume that each client has, at most, a mortgage quote.

Moreover, we can define the following views.

```
% hasMortgage(Name)
hasMortgage(N):- ex(Q,mortgageQuote(N,Q)).
```

A debtor is a client who has a past due with an amount greater than his balance.

```
% debtor(Name)
debtor(N):-
  client(N,B,S),
  pastDue(N,A),
  A>B.
```

The interest rate that is applicable to a client is specified by the next relation:

```
% interestRate(Name, Rate)
interestRate(N,2):-
  client(N,B,S),
  B<1200.

interestRate(N,5):-
  client(N,B,S),
  B>=1200.
```

The next relation specifies that a non-debtor client can be given a new mortgage in two situations. First, if he has no mortgage, a mortgage quote smaller than the 40% of his salary can be given. And, second, if he has a mortgage quote already, then the sum of this quote and the new one has to be smaller than that percentage.

```
% newMortgage(Name, Quote)
newMortgage(N,Q) :-
  client(N,B,S),
  not(debtor(N)),
  not(hasMortgage(N,Q1)),
  Q<=0.4*S.

newMortgage(N,Q) :-
  client(N,B,S),
  not(debtor(N)),
  mortgageQuote(N,Q2),
  Q+Q2<=0.4*S.

% getMortgage(Name)
getMortgage(N):- ex(Q,newMortgage(N,Q)).
```

If the client satisfies the requirements to be given a new mortgage, then it is possible to apply for a personal credit, whose amount is smaller than 6000. Otherwise, if the client does not satisfy that requirements, the amount must be between 6000 and 20000.

```
% personalCredit(Name, Amount)
personalCredit(N,A) :-
  (getMortgage(N),
   A<6000)
  ;
  (not(getMortgage(N)),
   A>=6000,A<20000).
```

For this database, we can query whether every client is a debtor:

```
fa(N,debtor(N)).
```

The answer is `false`.

Moreover it is possible to ask, for example, the quote and the salary of clients whose mortgage quote is greater than 100 with the next query:

```
ex(B,client(N,B,S),mortgageQuote(N,Q),(Q>=100)).
```

The answer constraint, that provides such information is the following disjunction:

```
(Q=400, S=1500, N=brown);
(Q=100, S=3000, N=mcandrew).
```

For knowing whether there are debtors with a past due amount greater than 1000, the following query can be formulated:

```
ex(N,ex(A,(debtor(N),pastDue(N,A),(A>1000)))).
```

and the answer is `true`. Note that we are using quantifiers for `N` and `A`, meaning that there are no explicit conditions over them. Otherwise, the answer will be a constraint over them.

The next query corresponds to the question: if for a client we assume that has a balance greater than 2000, what would the interest rate be?

```
fa(N,ex(S,ex(B,(client(N,B,S) =>
                    B>2000 => interestRate(N,R))))).
```

the answer is the constraint `R=5`. We are using nested implication to formulate hypothetical queries, in which we can assume both facts and constraints.

The next query involves negation and represents which clients can get a mortgage quote of 400 but *not* a personal credit.

```
newMortgage(N,400), not(personalCredit(N,A)).
```

And the answer is the constraint:

```
(N=mcandrew, A>=6000, A<20000)
```

This constraint means that it is possible to give a new mortgage to client McAndrew but it is *not* possible to give him a personal credit of an amount between 6000 and 20000. □

In this paper, we present an implementation of the fixpoint semantics presented in [Nieva et al. 2008], which is independent of the concrete constraint system. Also, we use a type system for identifying the constraint system to which each constraint in a database belongs. We propose several constraint systems as instances of $HH_{\neg}(\mathcal{C})$ and their solvers. And we explain how they are implemented.

The semantics of a database is computed as a set of pairs (A, C), where A is an atom and C a constraint, that can be deduced from both the extensional and intensional parts of the database. A can be understood as a n-ary relation instance, where their arguments are constrained by C. These pairs are computed by strata, classifying predicates by strata with a new form of stratification driven by both negations and implications occurring in rules. Each stratum should become saturated before trying to saturate any other higher stratum. However, as an implication may occur in a goal, the computation must take into account that the database is augmented with the hypothesis posed in the implication antecedent. Therefore,

a fixpoint computation has to be started from scratch since new pairs may be added at lower strata. So-nested subcomputations add a new complexity level with respect to usual bottom-up computations in deductive databases without implications.

Another complexity source comes again from implications, since the variables in $D \Rightarrow G$ can occur both in D and G. When a database Δ is augmented with the local clause D, those variables must be distinguished from other instances of the same variables in Δ. To this end, we recourse to Prolog attributed variables to identify them.

Finally, in order to find a stratification for ensuring finiteness of computations, a new dependency graph is described using a mutually recursive definition between the dependencies introduced by goals and clauses.

The rest of the paper is organized as follows. Section 2 recalls syntactical notions, the stratification needed for classifying predicates into strata due to both negation and implication, as well as stratified interpretations and the forcing relation. Section 3 introduces a user-oriented description of the system and the computation stages of the implementation. Section 4 describes the type system, constraint systems, their solvers and how they are implemented. Section 5 explains how the fixpoint semantics has been implemented by successive applications of an operator, which in turn implements the forcing relation of $HH_{\neg}(\mathcal{C})$. Section 6 describes a new form of the dependency graph needed to implement the forcing of the implication. Section 7 shows an actual, running example of the system in its current form. Section 8 summarizes some conclusions and sketches some future work.

2. Preliminaries

Here, we recall the foundations, presented in [Nieva et al. 2008], in which the implementation is based on.

2.1 Syntax

We consider a set of *defined predicate symbols*, representing the names of database relations, to build atoms, denoted by A, and *non-defined (built-in) predicate symbols*, including at least the equality predicate symbol $\approx$, to build constraints, denoted by C. We will also assume the existence of a set of constant and operator symbols, and a set of variables to build terms, denoted by t.

The constraints we consider belong to a generic system $\mathcal{C} = \langle \mathcal{L}_\mathcal{C}, \vdash_\mathcal{C} \rangle$, where $\mathcal{L}_\mathcal{C}$ is the constraint language and $\vdash_\mathcal{C}$ is a binary *entailment relation*. $\Gamma \vdash_\mathcal{C} C$ denotes that the constraint C is inferred in the constraint system $\mathcal{C}$ from the set of constraints Γ. Some minimal conditions are imposed on $\mathcal{C}$ to be a constraint system (see [Leach et al. 2001] for details). In particular, $\mathcal{C}$ is required to contain $\top$ (true) and $\bot$ (false), and to deal with $\wedge$, $\neg$, and the existential quantifier $\exists$; the constraint system has the responsibility of checking the satisfiability of answers in the constraint domain.

We say that a constraint C is $\mathcal{C}$-satisfiable if $\emptyset \vdash_\mathcal{C} \overline{\exists} C$, where $\overline{\exists} C$ stands for the existential closure of C. C and C' are $\mathcal{C}$-equivalent if $C \vdash_\mathcal{C} C'$ and $C' \vdash_\mathcal{C} C$.

The well-formed formulas in $HH_{\neg}(\mathcal{C})$ can be classified into clauses D (defining database relations) and goals (or queries) G. They are recursively defined by the following rules:

$$D ::= A \mid G \Rightarrow A \mid D_1 \wedge D_2 \mid \forall x D$$
$$G ::= A \mid \neg A \mid C \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \Rightarrow G \mid C \Rightarrow G \mid \exists x G \mid \forall x G$$

The programs, denoted by Δ, are sets of clauses and represent databases. Any Δ can always be given as an equivalent set, $elab(\Delta)$, of implicative clauses with atomic heads in the way we precise now. The *elaboration* of a program Δ is the set $elab(\Delta) = \bigcup_{D \in \Delta} elab(D)$, where $elab(D)$ is defined by:

$$elab(A) = \{\top \Rightarrow A\},$$
$$elab(D_1 \wedge D_2) = elab(D_1) \cup elab(D_2),$$
$$elab(G \Rightarrow A) = \{G \Rightarrow A\},$$
$$elab(\forall x D) = \{\forall x D' \mid D' \in elab(D)\}.$$

2.2 Stratification

The notion of stratification is used as a syntactical criterion to determine if a query to a database can be potentially be computed in a finite number of steps. The idea is that when $\neg A$ is going to be proved, the stratum of A has been previously saturated (all the answers for A are available) and $\neg A$ can be correctly computed. A stratification for a database is based on the construction of a dependency graph for a set of formulas [Zaniolo et al. 1997].

The nodes of the graph are the defined predicate symbols of the set. An implication of the form $F_1 \Rightarrow F_2$ produces edges and/or paths in the graph from the defined predicate symbols inside F_1 to each defined predicate symbol inside F_2. An edge will be negatively labeled when the corresponding atom occurs negated on the left side of the implication. Notice that in $HH_{\neg}(\mathcal{C})$ implications may occur not only between the head and the body of a clause, but also inside the goals, and therefore in the body of any clause. Since constraints do not include defined predicate symbols, they do not produce dependencies.

Those two kinds of edges are sufficient to guarantee the consistency of the following theory. However, in the implementation, an additional case of producing a negatively labeled edge will be considered. This new case will be explained in Section 6, after motivating it in Section 5.4.

DEFINITION 1. *Given a set of formulas Φ, its corresponding dependency graph DG_Φ, and two predicates p and q, we say that*

- q depends *on p if there is a path from p to q in DG_Φ.*
- q negatively depends *on p if there is a path from p to q in DG_Φ with at least one negatively labeled edge.*

Let $P = \{p_1, \ldots, p_n\}$ be the set of defined predicate symbols of Φ. A stratification *of Φ is a mapping $s : P \to \{1, \ldots, n\}$, such that $s(p) \leq s(q)$ if q depends on p, and $s(p) < s(q)$ if q negatively depends on p. Φ is stratifiable if there is a stratification for it.*

The stratum of a formula F, denoted by $str(F)$, is the maximum $s(p)$, where p is in the set of predicate symbols occurring in F.

Figure 1 shows the dependency graph for the bank database of the introduction. Negative edges are labelled with $\neg$.

2.3 Stratified Interpretations and Forcing Relation

Let $\mathcal{W}$ be the set of stratifiable databases Δ, with respect to the same fixed stratification s, At be the set of open atoms, and $\mathcal{SL}_\mathcal{C}$ be the set of $\mathcal{C}$-satisfiable constraints modulo $\mathcal{C}$-equivalence. Interpretations are classified on strata and each interpretation gives information up to its corresponding stratum.

DEFINITION 2. *Let $i \geq 1$, an* interpretation I over the stratum i *is a function $I : \mathcal{W} \to \mathcal{P}(At \times \mathcal{SL}_\mathcal{C})$, such that for any $\Delta \in \mathcal{W}$, and any $j > i$, $[I(\Delta)]_j = \emptyset$, where*

$$[I(\Delta)]_i = \{(A, C) \in I(\Delta) \mid str(A) = i\}.$$

We denote by $\mathcal{I}_i$ the set of interpretations over i.

For every $i \geq 1$, an order on $\mathcal{I}_i$ can be defined.

DEFINITION 3. *Let $i \geq 1$ and $I_1, I_2 \in \mathcal{I}_i$, I_1* is less or equal than I_2 at stratum i, *denoted by $I_1 \sqsubseteq_i I_2$, if for each $\Delta \in \mathcal{W}$ the following conditions are satisfied:*

- $[I_1(\Delta)]_j = [I_2(\Delta)]_j$, *for every* $1 \leq j < i$.
- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$.

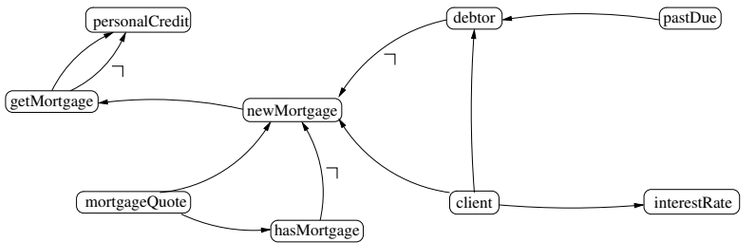


Figure 1. Dependency Graph for Example 1

For every $i \geq 1$, every chain of interpretations of $(\mathcal{I}_i, \sqsubseteq_i)$, $\{I_n\}_{n\geq 0}$, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$, has a least upper bound $\bigsqcup_{n\geq 0} I_n$, which can be defined as:

$$(\bigsqcup_{n\geq 0} I_n)(\Delta) = \bigcup_{n\geq 0} \{I_n(\Delta)\},$$

for any $\Delta \in \mathcal{W}$.

The following definition formalizes what means that a query G is *true* for an interpretation I in the context of a database Δ, if the constraint C is satisfied.

DEFINITION 4. *Let $i \geq 1$. The* forcing relation $\Vdash$ *between pairs I, Δ and pairs (G, C) (where $I \in \mathcal{I}_i$, $str(G) \leq i$, and C is $\mathcal{C}$-satisfiable) is recursively defined by the rules below.*

- $I, \Delta \Vdash (C', C) \iff C \vdash_{\mathcal{C}} C'$.
- $I, \Delta \Vdash (A, C) \iff (A, C) \in I(\Delta)$.
- $I, \Delta \Vdash (\neg A, C) \iff$ *for every* $(A, C') \in I(\Delta)$, $C \vdash_{\mathcal{C}} \neg C'$ *holds. If there is no pair of the form* (A, C') *in* $I(\Delta)$, *then* $C \equiv \top$.
- $I, \Delta \Vdash (G_1 \wedge G_2, C) \iff$ *for each* $i \in \{1, 2\}$, $I, \Delta \Vdash (G_i, C)$.
- $I, \Delta \Vdash (G_1 \vee G_2, C) \iff$ *for some* $i \in \{1, 2\}$ $I, \Delta \Vdash (G_i, C)$.
- $I, \Delta \Vdash (D \Rightarrow G, C) \iff I, \Delta \cup \{D\} \Vdash (G, C)$.
- $I, \Delta \Vdash (C' \Rightarrow G, C) \iff I, \Delta \Vdash (G, C \wedge C')$.
- $I, \Delta \Vdash (\exists x G, C) \iff$ *there is* C' *such that* $I, \Delta \Vdash (G[y/x], C')$, *where* y *does not occur free in* Δ, $\exists x G$, C, *and* $C \vdash_{\mathcal{C}} \exists y C'$.
- $I, \Delta \Vdash (\forall x G, C) \iff I, \Delta \Vdash (G[y/x], C)$, y *does not occur free in* Δ, $\forall x G$, C.

When $I, \Delta \Vdash (G, C)$, it is said that (G, C) is forced *by I, Δ.*

2.4 Fixpoint Semantics

The notion of truth at each stratum is given by means of the fixpoint of a continuous operator (for every stratum) transforming interpretations.

DEFINITION 5. *Let $i \geq 1$ represent a stratum. The* operator $T_i : \mathcal{I}_i \longrightarrow \mathcal{I}_i$ *transforms interpretations over i as follows. Let $I \in \mathcal{I}_i$, $\Delta \in \mathcal{W}$, and $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, then $(A, C) \in T_i(I)(\Delta)$ when:*

- $(A, C) \in [I(\Delta)]_j$ *for some $j < i$ or*
- $s(A) = i$ *and there is a variant* $\forall \overline{x}(G \Rightarrow A')$ *of a clause in* $elab(\Delta)$, *such that the variables $\overline{x}$ do not occur free in A, and* $I, \Delta \Vdash (\exists \overline{x}(A \approx A' \wedge G), C)$.

The operator T_1 has a least fixpoint $fix_1 = \bigsqcup_{n\geq 0} T_1^n(I_\perp)$, where the interpretation $I_\perp$ represents the constant function $\emptyset$. Proceeding successively on the same way, a chain:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots$$
$$\ldots \sqsubseteq_i T_i^n(fix_{i-1}) \sqsubseteq_i \ldots$$

can be defined for any stratum $i > 1$, and a fixpoint of it,

$$fix_i = \bigsqcup_{n\geq 0} T_i^n(fix_{i-1}),$$

can be found. In particular, if k is the maximum stratum in Δ, we simplify fix_k writing fix. Then, $fix(\Delta)$ represents the pairs (A, C) such that A can be deduced from Δ if C is satisfied.

3. System Description

In this section, we briefly introduce a user-oriented description of the system and the computation stages of the implementation.

The system incorporates the predefined data types `bool` (with `true` and `false` as elements) and `real`, an infinite data type, whose real numeric range is system-dependent. As well, the user is able to define new enumerated data types. A data type declaration is written as:

```
domain(data_type, [constant_1, ..., constant_n]).
```

Intervals for integers are allowed in data type declarations, as in:

```
domain(months, 1..12).
```

An n-arity predicate type declaration is written as:

```
type(predicate(type_1, ..., type_n)).
```

For instance, `type(client(client_dt, real))` is a type declaration, where `client_dt` can be defined as:

```
domain(client_dt, [smith, brown, mcandrew]).
```

The syntax for clauses is essentially as introduced in examples of Section 1, except for constraints, for which we use the syntax `constr(Dom,C)`, denoting a constraint `C` ranging over the domain `Dom`.

When, in the context of a database Δ, a user query `Q` is posed at the system prompt, it is translated into a clause `D ≡ A :- Q`, where `A` is an atom whose predicate symbol is `query` and whose arguments are the free variables in `Q` (they are implicitly existentially quantified in `Q` and universally quantified in `D`). In addition, the types for `query` are inferred and provided as the type declaration `type(query(Types))`.

Solving this query entails to add `D` to the current database Δ, i.e., to consider $\Delta' = \Delta \cup \{D\}$ for the following computation stages: 1) Check and infer predicate types; 2) Build the dependency graph of Δ'; 3) Compute a stratification for Δ' if there is any. If it is not stratifiable the system throws an error message an stops; 4) If the

previous step success, compute $fix(\Delta')$. The answer constraint to the query `Q` is the constraint `C` such that `(A,C)`$\in fix(\Delta')$.

Next, we describe the different components of the implementation in detail.

4. Implementing Constraint Solving

This section focuses on the implementation of constraint solving for the following particular constraint systems: Real numbers, integers, Boolean and user-defined enumerated types. Firstly, we comment on the type system needed to identify the types of variables which are used to send a constraint to its corresponding solver. Then, the constraint systems are described, including their predefined data values, functions and operators. Finally, we show the implementation of the constraint solvers, which makes use of SWI-Prolog [Wielemaker 2009] underlying constraint solvers.

4.1 Types

We have implemented a type checking and inferrer system for $HH_{\neg}(\mathcal{C})$ programs which is able to detect type inconsistencies and lack of type declarations, and to infer types for user queries. Types are locally annotated for each predicate symbol. A type annotation consists of storing the type of a variable in an attribute of this variable (cf. attributed variables [Holzbaur 1990]). A type is known in the context of a set of clauses: either a) an atom provides its type (i.e., because of its corresponding predicate type), or b) a constraint `constr(Dom,C)` provides its type. A type-conflict exception is raised when different types are tried to be assigned to the same variable. A lack-of-type-declaration exception is raised when no type is assigned to a variable.

4.2 Constraint Systems

As introduced, a constraint system provides a constraint language for expressing constraints and an entailment relation for ensuring satisfiability of constraints (this relation will be covered in the next subsection). Our constraint systems include the concrete syntax for the required values, symbols, connectives, and quantifiers as follows: "`true`", "`false`", "`=`", "`,`", "`not`" and "`ex(X,C)`" which represent, respectively, $\top$, $\bot$, $\approx$, $\wedge$, $\neg$ and $\exists$`X C`. In addition, we also include "`;`" for $\vee$ and "`/=`" for the negation of $\approx$.

We have proposed three constraint systems for the scheme $HH_{\neg}(\mathcal{C})$: Boolean, Reals, and Finite Domains. The first one consists of just the required components plus the universal quantifier. The constraint system Reals includes the type `real` (infinite set of real numeric values) and real constraint operators (`+`, `-`, `*`, ...) and functions (`abs`, `sin`, `exp`, `min`, ...).

Finite Domains represent a family of specific constraint systems ranging over denumerable sets. Enumerated types are included as well as (finite) integer numeric types. Whereas the constraint systems Boolean and Reals have attached predefined types, Finite Domains do not. This system also includes comparison operators (`>`, `>=`, ...), universally quantified constraints (`fa(X,C)`, as above), and the domain constraint `X in Range`, where `Range` is a subset of data values built with `V1..V2`, which denotes the set of values in the closed interval between `V1` and `V2`, and `R1\/R2`, which denotes the union of ranges. A finite domain may also include constraint operators (as `+`, `-`, ...) and constraint functions (as `abs`, `min`, ...). Note that relevant primitive functions for each system should be clear from their intended semantics (`+` might not be relevant for Booleans, although it can be used). We allow to use the same symbols to build constraints in different systems; for instance, both `constr(real, X>Y)` and `constr(month, X>Y)` make sense in their respective constraint systems.

4.3 Constraint Solvers

We have considered the entailment relation of the classical logic for every constraint system. This entailment satisfies the minimal condition imposed to constraint systems. For implementing this relation, we provide a constraint solver with a generic interface `solve(C,SC)` for `C` $\vdash_{\mathcal{C}}$ `SC`, intended to solve a constraint `C`, check its satisfiability and produce a *solved* form `SC`. A solved form `SC` corresponding to a constraint `C` is a simplified, more readable form of the constraint wrt. `C`. A solved form can be a disjunction of simple constraints, where a simple constraint does neither include disjunctions nor quantifications, nor negations. This generic interface is implemented as follows:

```
solve(C,SC) :-
  partition_ctr(C,DCs),
  solve_ctr_list(DCs,SDCs),
  ctr_list_to_ctr(SDCs,CC),
  simplify_ctr(CC,SC).
```

Its first call partitions the input constraint into a list whose components belong to different constraint domains. The next call posts each component to its corresponding solve as a call to the predicate `solveFD` (described later). After, the solved constraint represented as a list is transformed back into a constraint data structure. Finally, this constraint is simplified by logical axioms as De Morgan's laws.

In addition to the generic interface, the particular interface `solve(Dom,C,SC)` is also provided, which is useful when the domain `Dom` is already known and can be directly posted to its corresponding solver.

Next, we describe our implementation of the constraint solvers for the constraint systems we propose as practical instances of $HH_{\neg}(\mathcal{C})$.

We rely on the underlying constraint solvers already available in SWI-Prolog [Wielemaker 2009] for implementing the constraint systems Finite Domains, Boolean and Reals. For certain constraints, we are able to map them to constraints in the underlying SWI-Prolog finite domain solver because we map data values to integers. Before posting to this solver, a constraint is rewritten with the mapped integer values and, after solving, the solved constraint is rewritten back with the corresponding enumerated values. On the other hand, there are constraints that the underlying solvers cannot directly handle (quantifiers and disjunctions) which we explicitly handle as will be shown later. Since SWI- Prolog does not provide a Boolean solver, we resort to the finite domain constraint solver for solving Boolean constraints, and provide the predefined constraint system `bool` which is handled as any other enumerated constraint system.

For the solvers of the constraint systems Finite Domains and Boolean, the following predicates are available:

- `solveFD(+Domain,+Constraint,-SolvedConstraint)`
  It solves the input `Constraint` over `Domain` and returns its solved form `SolvedConstraint` associated to `Domain` , if it is satisfiable.
- `constr_conjFD(+Domain,-C1,+C2,+C)`
  It is read as "`C1,C2 = C`", and computes the component `C1` of the conjunction `C` under the given domain.

Since we consider classical logic for these particular constraint systems, the following implementation for the second predicate is sound:

```
constr_conjFD(Domain,C1,C2,C) :-
  solveFD(Domain,(not(C2);C),C1),
  solveFD(Domain,(C1,C2),SC).
```

Whilst the second line is intended to compute `C1` under the assumption of success, the following lines check that the computed constraint is satisfiable.

The code excerpt of Figure 2 implements the required behaviour:

Note that line `(05)` is intended to replace quantified variables by fresh ones in order to avoid a name clash. Line `(07)` maps domain data values with integers, whereas line `(16)` replaces back the (integer) computed data values by the corresponding, mapped data values. The core of constraint solving lays between lines `(09)-(11)`, where, first, the constraint is tried to be solved (see next paragraph describing the predicate `solveFD_ctr`). Second, it is checked for satisfiability, that is, trying to find a single, concrete solution via labeling. And, third, the underlying constraint store is projected with respect to the relevant variables (i.e., those occurring in the input constraint plus the possible new ones computed by the underlying solver). Lines `(13)-(15)` are simply intended for data structure formatting.

Next, we describe the predicate:

`solveFD_ctr(+Constraint,-Satisfiable)`,

which receives a constraint and returns whether it is satisfiable or not. The first case of this predicate corresponds to a constraint supported by the constraint solver of SWI-Prolog (where `#>` is the finite domain constraint comparison operator provided by this solver):

```
solveFD_ctr(X>Y,true) :-
  !,
  X#>Y.
```

Negation is, as shown below, explicitly handled because it can apply to unsupported constraints. The predicate

`complement(+Constraint,-ComplementedConstraint)`

computes the complemented constraint before solving it.

```
solveFD_ctr(not(C),B) :-
  !,
  complement(C,NotC),
  solveFD_ctr(NotC,B).
```

An example of handling unsupported constraints is disjunction, which is computed by collecting all answers (cf. line `(08)`). Solving this constraint is as follows:

```
solveFD_ctr((C1;_C2),true) :-
  solveFD_ctr(C1,true).

solveFD_ctr((_C1;C2),true) :-
  solveFD_ctr(C2,true).
```

Finally, we describe quantifiers. Firstly, the existential quantifier is implemented as follows, where in the last but one line `satisfiable(FC,true)` tries to find a concrete value satisfying `FC`:

```
solveFD_ctr(ex(X,C),B) :-
  !,
  % Replace X by a fresh variable _FX in C:
  swap(X,_FX,C,FC),
  get_current_domain(DN),
  constrain_domains(FC,DN),
  % Solving:
  (solveFD_ctr(FC,true),
   % Checking satisfiability:
   satisfiable(FC,true),
   B=true ; B=false).
```

The universal quantifier is solved by imposing a conjunctive constraint `C` for all the values of `X` in the solving domain (cf. the call to `solve_forall`):

```
solveFD_ctr(fa(X,C),B) :-
  !,
  get_current_domain(Domain),
  domain_bounds(Domain,L,U),
  (solve_forall(X,C,L,U) ->
    B=true
   ;
    B=false).
```

The constraint solver for Reals follows a similar but simpler route for its implementation since there are neither universal quantifiers, nor domain data values to map.

5. Implementing the Fixpoint Semantics

In this section, we present the implementation of the core system, which is independent from the concrete constrain systems explained in the previous section.

5.1 Fixpoint by Strata

For the fixpoint computation we assume a stratified database Δ, i.e., a partition $st_1, \ldots, st_k$ over the predicate symbols defined in it (the stratification algorithm will be explained in Section 6). A clause of the form `A :- G` is interpreted as $\forall X_1, \ldots, X_n(\mathtt{G} \Rightarrow \mathtt{A})$, being $X_1, \ldots, X_n$ the free variables of $(\mathtt{A}, \mathtt{G})$, and is encoded as the Prolog term

`rule(St,Vars,A,G)`

where `St` $= str($`A`$)$ and `Vars`$= [X_1, \ldots, X_n]$.

The fixpoint is computed stratum by stratum (although a stratum may require the computation of the fixpoint for a previous stratum when the program is enlarged due to implications as we will see in Section 5.4). The predicate

`fixPointStrat(+Delta, +St, -Fix)`

computes `Fix` $= fix_{\mathtt{St}}($`Delta`$)$. Then, if `Delta` represents a database such that `St` $= str($`Delta`$) = k$, this predicate gives $fix_k($`Delta`$)$, computing previous fixpoints from `St` $= 0$ to `St` $= k$.

```
fixPointStrat(_Delta,0,[]) :- !.

fixPointStrat(Delta,St,FixSt) :- St1 is St-1,
  fixPointStrat(Delta,St1,FixSt1),
  iterT(Delta,St,FixSt1,FixSt).
```

Each fixpoint is evaluated by iterating the fixpoint operator as follows:

```
iterT(Delta,St,I,FixSt) :-
  opT(Delta,Delta,St,I,TI),
  (
    I==TI,!, FixSt=I
  ;
    iterT(Delta,St,TI,FixSt) ).
```

`I` represents the current computed interpretation and `FixSt` will be the fixpoint for the stratum under consideration. The operator is iterated until no more information can be added to the interpretation (`I==TI`), i.e., we have reached the fixpoint for the stratum `St`. The predicate `opT` is detailed below.

```
(00) solveFD(DN,C,SC) :-
(01)   set_current_domain(DN),          % A flag storing the current domain
(02)   copy_term(C,FC),                 % Input variables keep untouched
(03)   get_vars(C,Vars),                % Input variables are held to be
(04)   get_vars(FC,FVars),              %  mapped to the solved new vars
(05)   swap_qvars_by_fvars(FC,QFC),     % Replace quantified vars by fresh ones
(06)   constrain_domains(QFC,DN),       % Constrain variables to the current domain
(07)   domain_to_int(QFC,DN,IC),        % Domain mapping from enumerated to integer
(08)   bagof((FVars,Cs,Sat),            % (Fresh vars,Constraints,Satisfiable)
(09)     (solveFD_ctr(IC,true),         % Solving
(10)      satisfiable(IC,Sat),          % Check satisfiability
(11)      project_ctrs(FVars,Vars,Cs)   % Project constraints wrt. input vars
(12)        ), LFVarsCsS), !            % List of (Fresh vars,Constraints,Satisfiable)
(13)   filter_ctr_list(LFVarsCsS,LICs), % Pick solved constraints
(14)   simplify_disj_list(LICs,SLICs),  % Simplify the disjunctive list
(15)   disj_list_to_ctr(SLICs,ISC),     % Convert list to constraint
(16)   int_to_domain(ISC,DN,SC).        % Map domain from integer to enumerated
```

Figure 2. The Predicate `solveFD` for solving Finite Domain Constraints

5.2 Fixpoint Operator

The predicate `opT` corresponds to the application of the operator T_i (for some stratum i) to a given interpretation. Following Definition 5, the predicate

```
opT(+Rules,+Delta,+St,+I,-TI)
```

receives in `I` the set of pairs of $T_i^n(fix_{i-1})$(`Delta`) for some $n \geq 0$, the stratum $i =$ `St` and computes `TI` $= T_i^{n+1}(fix_{i-1})$(`Delta`). The call to `opT` from `iterT` has the form

```
opT(Delta,Delta,St,I,TI)
```

taking `Delta` twice because it uses each clause of `Delta` separately, but the forcing relation will need the full database `Delta`. This operator only uses the clauses of the current stratum `St` (second clause) and skips the rest (last clause).

```
opT([],_Delta,_St,I,I).

opT([rule(St,Vars,A,G)|Rs],Delta,St,I,TI) :-
  !,
  rename(Vars,(A,G),Vars1,(A1,G1)),
  flatHead(A1,A2,Cs),
  buildExists(Vars1,(Cs,G1),G2),
  (
    force(Delta,I,G2,C), !,
    addItemLst([(A2,C)],I,I1)
  ;
    I1=I ),
  opT(Rs,Delta,St,I1,TI).

opT([_|Rs],Delta,St,I,I1) :-
  opT(Rs,Delta,St,I,I1).
```

The second clause performs some initial transformations on the rule `rule(St,Vars,A,G)`: the predicates `rename`, `flatHead` and `buildExists` build the goal to be forced

$$\texttt{G2} = \exists\, \texttt{Vars1}\ (\texttt{G1} \wedge \texttt{A1} \approx \texttt{A2}),$$

being $\forall$ `Vars1` (`G1` $\Rightarrow$ `A1`) a variant of `rule(St,Vars,A,G)`. Then it tries to force the obtained goal `G2` using `Delta` and the current interpretation `I`. If it succeeds, we obtain the associated constraint `C` and we add the pair `(A2,C)` to such an interpretation. Finally, `opT` performs the same operation on the rest of rules `Rs`.

5.3 Forcing Relation

We implement the forcing relation $\models$ of Definition 4 by means of the predicate

```
force(+Delta,+I,+G,-C).
```

Given `I` $= T_i^n(fix_{i-1})$(`Delta`) for some $n \geq 0$ and a fixed stratum $i > 0$, `force` is successful if $T_i^n(fix_{i-1})$, `Delta` $\models$ (`G`, `C`). An important point to understand the implementation is to keep in mind the deterministic nature of this predicate. The definition of $\models$ establishes conditions on a constraint C in order to satisfy `I`, `Delta` $\models$ (`G`,C), but the predicate `force` must build a concrete constraint `C`. In addition, each possible answer constraint for a goal must be captured in a single answer constraint (possibly) using disjunctions. There is a clause of `force` for each goal structure. We explain them shortly, except for the case of implication, that will be studied in the next subsection:

```
force(_Delta,_I,constr(Dom,C),C1) :-
  !, solve(Dom,C,C1).

force(Delta,I,(G1,G2),C) :-
  !, force(Delta,I,G1,C1),
  force(Delta,I,G2,C2),
  solve((C1,C2),C).

force(Delta,I,(G1;G2),C) :- !,
  ( force(Delta,I,G1,C1), !,
      ( force(Delta,I,G2,C2), !,
        solve((C1;C2),C)
      ;
        solve(C1,C) )
  ;
    force(Delta,I,G2,C2),
    solve(C2,C) ).

force(Delta,I,(constr(Dom,C)=>G),C2) :-
  !, force(Delta,I,G,C1),
  constr_conj(Dom,C2,C,C1).

force(Delta,I,ex(X,G),C) :-
  !, replace(X,X1,G,G1),
  force(Delta,I,G1,C1),
  solve(ex(X1,C1),C).
```

```
force(Delta,I,fa(X,G),C) :-
  !, replace(X,X1,G,G1),
  force(Delta,I,G1,C1),
  solve(fa(X1,C1),C).

force(_Delta,I,not(At),C) :-
  !, lookUpAll(At,I,Ls),
  ( Ls==[], !, C=true
  ;
    buildNegConj(Ls,NLs),
    solve(NLs,C) ).

force(_Delta,I,At,C) :-
  !, lookUpAll(At,I,Cs),
  buildDisj(Cs,C1),
  solve(C1,C).
```

The first clause stands for the forcing of a constraint `C` within a domain `Dom`, that is processed by calling the constraint solver. The second stands for a conjunction `G1,G2`; it forces both goals, and then solves the conjunction of the resulting answer constraints. For a disjunction `G1;G2` (third clause) there are four possible (and exclusive) situations: both goals can be forced, only `G1`, only `G2`, or neither of two; the answer constraint is obtained by solving the corresponding constraints or failing in the last case. The fourth clause of `force` corresponds to an implication with a constraint as antecedent; in this case the predicate `constr_conj` obtains a constraint `C2` such that if `I` forces `(G,C1)` then the conjunction `C2,C` is equivalent to `C1`.

For the universal quantifier, according to the Definition 4, to find `C` such that $\mathtt{I},\mathtt{Delta} \models (\forall \mathtt{X}\ \mathtt{G},\mathtt{C})$, we obtain `G1` as the result of replacing `X` by a new variable `X1` in `G`; then we prove $\mathtt{I},\mathtt{Delta} \models (\mathtt{G1},\mathtt{C1})$ and finally `C` is obtained by solving $\forall \mathtt{X1}\ \mathtt{C1}$. For the existential quantifier, according to the Definition 4, we find `C` such that there is `C'` satisfying $\mathtt{I},\mathtt{Delta} \models (\mathtt{G}[\mathtt{X1}/\mathtt{X}],\mathtt{C}')$ and $\mathtt{C} \vdash_{\mathcal{C}} \exists \mathtt{X1}\ \mathtt{C}'$. Then we can use `C` as the solved form of $\exists \mathtt{X1}\ \mathtt{C}'$ in the implementation.

For negated atoms `not(At)`, thanks to the stratification we can ensure that every possible atom `At` deducible from the database is already present in the current interpretation `I`. Then, by means of `lookUpAll(At,I,Ls)` we find the list `Ls=[C1,...,Cn]` such that `(At,Ci)`$\in$`I`. The variable `NLs` is used to build the constraint $\neg\mathtt{C1} \wedge \ldots \wedge \neg\mathtt{Cn}$ (or `true` if `Ls=[]`), that we must solve to obtain the constraint `C` we are looking for.

The last (default) case is the forcing of an atom `At`. As before, we search for all the pairs `(At,C1),...,(At,Cn)`$\in$`I` and then we build the disjunction $\mathtt{C1} \vee \ldots \vee \mathtt{Cn}$ and solve it with `solve`.

5.4 The Case of `D => G` in the Forcing Relation

Implementing `force(Delta,I,(D=>G),C)` requires some special treatment. In this case, according with the definition of the relation $\models$ (see Definition 4), `Delta` is augmented with the clause `D`. Remains that the current set `I` has been computed in accordance with the database `Delta`, in such a way that if i and n are, respectively, the stratum and iteration under construction, $(A,C) \in \mathtt{I} \Leftrightarrow (A,C) \in T_i^n(I')(\mathtt{Delta})$, where I' is the fixpoint for the stratum $i-1$, built from `Delta`. According to the theory, the next step will be to prove $T_i^n(I'),\mathtt{Delta} \cup \{\mathtt{D}\} \models (\mathtt{G},\mathtt{C})$. But the question is how to compute $T_i^n(I')(\mathtt{Delta} \cup \{\mathtt{D}\})$. Notice that `I` is not useful here. First, because $I(\Delta) \subseteq I(\Delta \cup \{D\})$ does not hold for every I, Δ, D. Second, because `I` has been built considering always `Delta`, in particular the fixpoint I' has been computed for `Delta`, then it represents $fix_{i-1}(\mathtt{Delta})$. So nothing is known about the needed set $T_i^n(I')(\mathtt{Delta} \cup \{\mathtt{D}\})$.

What it is happening is that the definition of the fixpoint operator T_i is not constructive for the case of implication due to the increase of the set of clauses. To solve this obstacle, we have adopted a conservative position: to compute locally the fixpoint of the stratum j for `Delta` $\cup \{\mathtt{D}\}$, where j is the stratum of `G`, that means $fix_j(\mathtt{Delta} \cup \{\mathtt{D}\})$, and then prove if $fix_j, \mathtt{Delta} \cup \{\mathtt{D}\} \models (\mathtt{G},\mathtt{C})$.

Of course, the complexity of the algorithm is considerably augmented on this case. But the code keeps simple. The corresponding clause for the predicate `force` is as follows:

```
force(Delta,I,(D=>G),C) :-
  !,
  elab(D,De),
  localRules(De,Ls),
  getStrat(G,StG),
  addLocalRules(Ls,Delta,Delta1),
  fixPointStrat(Delta1,StG,Fix),
  force(Delta1,Fix,G,C).
```

Calling to `elab(D,De)`, `localRules(De,Ls)`, `getStrat(G,StG)` and `addLocalRules(Ls,Delta,Delta1)`, the elaboration of the set of clauses `Delta` $\cup \{\mathtt{D}\}$, is produced giving the corresponding set `Delta1` in the used format. The execution of

$$\mathtt{fixPointStrat}(\mathtt{Delta1},\mathtt{StG},\mathtt{Fix})$$

finds $\mathtt{Fix} = fix_j(\mathtt{Delta1})$, where $j = \mathtt{StG}$ is the stratum of `G`, the consequent of the initial goal `D => G`. Once `Fix` is computed, it is needed to force `G` with it and the augmented set `Delta1`. This corresponds to prove

$$\mathtt{force}(\mathtt{Delta1},\mathtt{Fix},\mathtt{G},\mathtt{C}),$$

that implies $T_i^n(I'),\mathtt{Delta} \cup \{\mathtt{D}\} \models (\mathtt{G},\mathtt{C})$, as we wanted to prove.

This solution causes the following problem. Consider a clause in `Delta` of the form `A :- D => G`, such that $i = str(\mathtt{A})$ and $j = str(\mathtt{G})$; from Definition 1, $j \leq i$ can be deduced. During the computation of $fix_i(\mathtt{Delta})$, the predicate `opT` takes this clause into account, in order to look for a pair `(A,C)` to be added to the current `I`. Then

$$\mathtt{force}(\mathtt{Delta},\mathtt{I},(\mathtt{D} => \mathtt{G}),\mathtt{C})$$

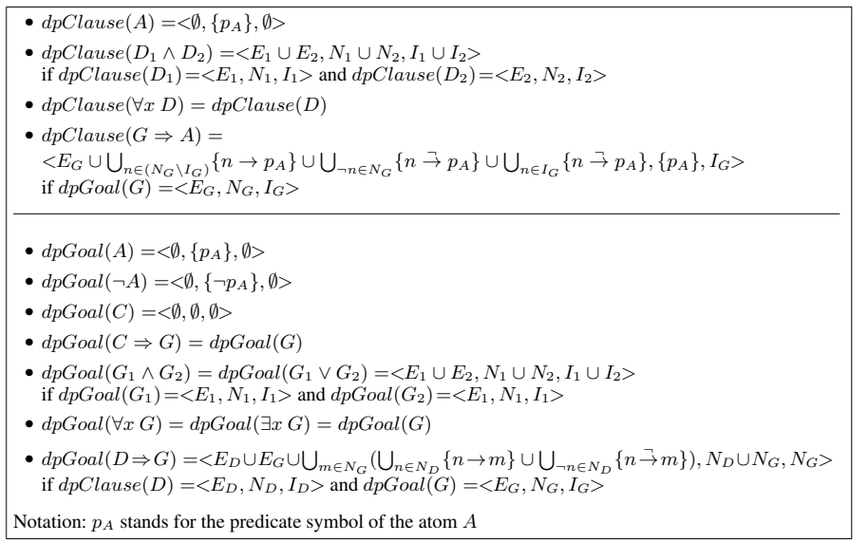
is executed which calls to

$$\mathtt{fixPointStrat}(\mathtt{Delta1},\mathtt{j},\mathtt{Fix}),$$

where $\mathtt{Delta1} = \mathtt{Delta} \cup \{\mathtt{D}\}$ (except elaboration and variable renaming). If $j = i$, that means to build $fix_i(\mathtt{Delta1})$, so the clause `A :- D => G` will be tried again, because the stratum of `A` is i. This gives rise to a non-terminating loop, since `Delta1` is augmented with the elaboration of `D` once more, and so on. However, if $j < i$, $\mathtt{Fix} = fix_j(\mathtt{Delta1})$ can be correctly built. This is the reason why, in the construction of dependency graphs, a new kind of negatively labeled edges has been incorporated, that ensures $str(\mathtt{G}) < str(\mathtt{A})$ in these cases. The details are explained in the following section.

6. Implementing the Dependency Graph

In [Nieva et al. 2006], we defined an algorithm to compute the dependency graph of any set of $HH_{\neg}(\mathcal{C})$ formulas. The main ideas and definitions are introduced in Section 2.2. Due to the problem introduced by nested implications, that we have exposed previously, a stronger definition of stratifiable database has been adopted in the current implementation. Now, these implications will introduce additional negative dependencies in the dependency graph. More precisely, if $G \Rightarrow A$ is a clause, such that G contains a subgoal of the form $D \Rightarrow G'$, this nested implication produces negatively labeled edges from the definite predicate symbols of G' to the predicate symbol of A.



- $dpClause(A) = <\emptyset, \{p_A\}, \emptyset>$
- $dpClause(D_1 \wedge D_2) = <E_1 \cup E_2, N_1 \cup N_2, I_1 \cup I_2>$
  if $dpClause(D_1) = <E_1, N_1, I_1>$ and $dpClause(D_2) = <E_2, N_2, I_2>$
- $dpClause(\forall x\ D) = dpClause(D)$
- $dpClause(G \Rightarrow A) =$
  $<E_G \cup \bigcup_{n \in (N_G \setminus I_G)} \{n \rightarrow p_A\} \cup \bigcup_{\neg n \in N_G} \{n \stackrel{\neg}{\rightarrow} p_A\} \cup \bigcup_{n \in I_G} \{n \stackrel{\neg}{\rightarrow} p_A\}, \{p_A\}, I_G>$
  if $dpGoal(G) = <E_G, N_G, I_G>$

- $dpGoal(A) = <\emptyset, \{p_A\}, \emptyset>$
- $dpGoal(\neg A) = <\emptyset, \{\neg p_A\}, \emptyset>$
- $dpGoal(C) = <\emptyset, \emptyset, \emptyset>$
- $dpGoal(C \Rightarrow G) = dpGoal(G)$
- $dpGoal(G_1 \wedge G_2) = dpGoal(G_1 \vee G_2) = <E_1 \cup E_2, N_1 \cup N_2, I_1 \cup I_2>$
  if $dpGoal(G_1) = <E_1, N_1, I_1>$ and $dpGoal(G_2) = <E_1, N_1, I_1>$
- $dpGoal(\forall x\ G) = dpGoal(\exists x\ G) = dpGoal(G)$
- $dpGoal(D \Rightarrow G) = <E_D \cup E_G \cup \bigcup_{m \in N_G} (\bigcup_{n \in N_D} \{n \rightarrow m\} \cup \bigcup_{\neg n \in N_D} \{n \stackrel{\neg}{\rightarrow} m\}), N_D \cup N_G, N_G>$
  if $dpClause(D) = <E_D, N_D, I_D>$ and $dpGoal(G) = <E_G, N_G, I_G>$

Notation: p_A stands for the predicate symbol of the atom A

Figure 3. Dependency Graph for Clauses and Goals

The algorithm for calculating the dependency graph is expressed by means of the mutually recursive functions $dpClause$ and $dpGoal$ defined in Figure 3, depending on the structure of the formula. Both they return a triple $<E, N, I>$, where E is a set of edges of the form $p \rightarrow q$ or $p \stackrel{\neg}{\rightarrow} q$, N and I are auxiliary sets of link-nodes. N is used to store information about the positive-negative predicates, and I stores the predicates involved in nested implications. Using the function $dpClause$ it is straightforward to calculate the dependency graph of a set of clauses as the union of the edges obtained for each element of the set. The dependency graph is used to define the stratification in $HH_{\neg}(\mathcal{C})$, that is a syntactic condition for ensuring finiteness in the computations with negated atoms.

EXAMPLE 2. Consider the clause:
$D \equiv \forall x(G \Rightarrow p(x))$, where
$G \equiv \exists y(q(x,y) \Rightarrow (r(x) \wedge s(y))) \wedge \neg t(x)$. Then
$dpGoal(G) =$
$<\{q \rightarrow r, q \rightarrow s\}, \{q, r, s, \neg t\}, \{r, s\}>$,
$dpClause(D) =$
$<\{q \rightarrow r, q \rightarrow s, q \rightarrow p, r \stackrel{\neg}{\rightarrow} p, s \stackrel{\neg}{\rightarrow} p, t \stackrel{\neg}{\rightarrow} p\}, \{p\}, \{r, s\}>$.
The first component of the tuple $dpClause(D)$ is the dependency graph associated to D. A database with just this clause is stratifiable, but if the clause:

$$D' \equiv \forall x \forall y(p(x) \Rightarrow q(x,y))$$

is also present, the database becomes non stratifiable. □

The concrete algorithm for finding a stratification for Δ (or for checking that it is not stratifiable) associates to each predicate symbol p an integer variable $X_p \in [1..N]$, where N is the number of predicate symbols of Δ, and generates an inequation system: each dependency $p \rightarrow q$ produces $X_p \leq X_q$ and $p \stackrel{\neg}{\rightarrow} q$ produces $X_p < X_q$. Then, solving this system (if possible) provides the stratum of each p in X_p. The stratification algorithm ends with a concrete stratification if there exists one or stops with an error message (in a polynomial time with respect to the number of predicate symbols in the database).

A stratification for the clause D of Example 2 will collect all the predicates in the stratum 1 except p, which will be in the stratum 2. In particular $X_q < X_p$. Intuitively, this means that for evaluating p, the rest of predicates should be evaluated before, in particular q, that takes part of a nested implication. If the previous clause D' is considered, we would also have $X_q \geq X_p$ and the inequation system does not have any solution.

The new negative dependencies introduced in the graph due to nested implications restrict the class of stratifiable programs, i.e., the syntax of our programs. Nevertheless, in practice this restriction does not means a loss of expressivity in the language, that is much more powerful than relational algebra or Datalog.

In the next section, we show (in Figure 4) the whole dependency graph associated to the bank database plus the queries of Example 1. This set is stratifiable. Notice that the edge `interestRate` $\stackrel{\neg}{\rightarrow}$ `query4` is due to the first nested implication inside the clause defining `query4`:

```
query4(R) :- fa(N,ex(S,ex(B,(client(N,B,S) =>
   constr(real,B>2000) => interestRate(N,R))))).
```

This implication produces also `client` $\rightarrow$ `interestRate` and `client` $\rightarrow$ `query4`. So, by transitivity, `query4` negatively depends on `interestRate`, but it also negatively depends on `client`, because `interestRate` depends on `client`.

7. A System Session

Next, we show the result of executing our system for the database and queries Δ that we have shown in Example 1. In this example, the following enumerated domain and types are declared:

```
domain(client_dt,[smith,brown,mcandrew]).
```

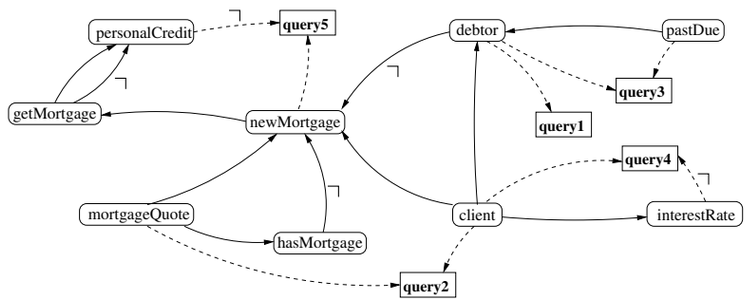


Figure 4. Dependency Graph for Example 1 with some queries.

```
type(client(client_dt,real,real)).
type(pastDue(client_dt,real)).
type(mortgageQuote(client_dt,real)).
type(hasMortgageQuote(client_dt)).
type(debtor(client_dt)).
type(interestRate(client_dt,real)).
type(newMortgage(client_dt,real)).
type(getMortgage(client_dt)).
type(personalCredit(client_dt,real)).
```

The following clauses corresponding to a number of queries are added to the bank database. They are shown along with their types, which are inferred in the context of the above declarations.

```
type(query1).
query1 :- fa(N,debtor(N)).

type(query2(client_dt,real,real)).
query2(N,S,Q) :-
  ex(B,client(N,B,S),mortgageQuote(N,Q),
  constr(real,Q>=100)).

type(query3).
query3 :-
  ex(N,ex(A,(debtor(N),pastDue(N,A),
  constr(real,A>1000)))).

type(query4(real)).
query4(R) :-
  fa(N,ex(S,ex(B,(client(N,B,S) =>
                constr(real,B>2000) =>
                interestRate(N,R))))).

type(query5(client_dt,real)).
query5(N,A) :-
  newMortgage(N,400), not(personalCredit(N,A)).
```

The dependency graph calculated for the current set of clauses is shown in Figure 4 (we use dashed lines for dependencies introduced by the queries).

From this graph, the stratification algorithm associates:

- Stratum 1 to `client`, `pastDue`, `mortgageQuote`, `debtor`, `interestRate`, `hasMortgage`, `query1`, `query2` and `query3`.
- Stratum 2 to `newMortgage`,`getMortgage`, and `query4`.
- Stratum 3 to `personalCredit`.
- Stratum 4 to `query5`.

Since Δ is stratifiable, the computation of

$$\texttt{fixPointStrat}(\Delta,4,\texttt{Fix})$$

begins calculating $fix_i(\Delta)$, stratum by stratum from $i = 1$ to 4, in order to obtain $\texttt{Fix} = fix_4(\Delta)$.

1. Computation of $fix_1(\Delta)$.

   The first iteration of T_1 over the empty set, that corresponds to the execution of $\texttt{opT}(\Delta,\Delta,1,[],\texttt{TI})$, obtains in `TI` the pairs associated to the extensional database:

   ```
   (client(smith,2000,1200), true),
   (client(brown,1000,1500), true),
   (client(mcandrew,5300,3000), true)
   (pastDue(smith,3000), true),
   (pastDue(mcandrew,100, true),
   (mortgageQuote(brown,400), true),
   (mortgageQuote(mcandrew,100), true)
   ```

   The fixpoint computation of this first stratum requires one more iteration of T_1. After this, the following pairs are added:

   ```
   (debtor(X), X=smith),
   (interestRate(smith, 2), true),
   (interestRate(X,Y),
      ((X=brown, Y=5);
       (X=mcandrew, Y=5))),
   (query2(X,Y,Z),
      ((Y=400, Z=1500, X=brown);
       (Y=100, Z=3000, X=mcandrew))),
   (query3, true),
   (hasMortgage(X), (X=brown;X=mcandrew))
   ```

   Note that no pair due to `query1` is added at this stage since the universally quantified constraint in this clause amounts to a conjunctive constraint over the domain of `debtor`, i.e., imposing that *all* the clients in `client_dt` are debtors, which is not the case.

2. Computation of $fix_2(\Delta)$.

   Determining whether a pair `(query4(X),C)` can be added to the current set of pairs gives to locally recalculate fix_1, but this time for $\Delta \cup \{\texttt{client}(\texttt{N},\texttt{B},\texttt{S})\}$.

   To obtain $fix_2(\Delta)$, in the first iteration and after the appropriate computations to calculate $fix_1(\Delta \cup \{\texttt{client}(\texttt{N},\texttt{B},\texttt{S})\})$, the following pairs are added to $fix_1(\Delta)$:

```
(query4(X), X=5),
(newMortgage(X,Y),
   ((Y=<200, X=brown);
    (Y=<1100, X=mcandrew)))
```

And, in the second iteration, the next pair is added:

```
(getMortgage(X), (X=brown;X=mcandrew))
```

3. Computation of $fix_3(\Delta)$.

   Here, a pair for the predicate `personalCredit` is added to the previous fixpoint:

```
(personalCredit(X,Y),
   ((Y>=6000,Y<20000, X=smith);
    (Y<6000, X=brown);
    (Y<6000, X=mcandrew)))
```

4. Computation of $fix_4(\Delta)$.

   The final fixpoint requires one iteration of T_4 over the fixpoint of the third stratum

$$\texttt{iterT}(\Delta,4,fix_3(\Delta),\texttt{FixSt}),$$

   obtaining the following new pair:

```
(query5(X,Y),
   (X=mcandrew, Y>=6000, Y=<20000))
```

   This completes the result, and $fix_4(\Delta) = \texttt{FixSt}$ captures the semantics of our database and queries.

In the example, the stratification and fixpoint have been calculated for the database together with all the queries we had formulated. Hence they can be seen as predefined views. It is not the case that the fixpoint should be recomputed each time a query is posed. A more reusable behaviour is also possible in many cases. For a database `Delta`, a stratification s and a fixpoint $\texttt{Fix} = fix(\texttt{Delta})$ can be computed and stored. If the stratification s is valid for the posed query `Q`, then the expected answer constraint `C` can be obtained by executing: `force(Delta,Fix,Q,C)`.

8. Conclusions and Future Work

In [Nieva et al. 2008] we presented a formalization of the constraint logic programing scheme $HH_\neg(\mathcal{C})$ as an expressive deductive database system that returns constraints as answer of the queries. A semantics was developed, following stratification and fixpoint techniques, usual in the framework of deductive database semantics. But the underlying logic of our system embraces both constraints and new connectives on the goals or queries (implications, negation and quantifiers). This fact enlarges expressivity and efficiency, but introduces some penalties in the implementation.

We have developed a prototype of a deductive database system that shows the feasibility of the fixpoint semantics as a base for an actual implementation. The core of this implementation is independent of the concrete constraint system. Several constraint systems are implemented as instances of this scheme. In particular, we have considered real numbers, integers, Booleans and user defined enumerated types (all of these, but reals, belong to the finite domain constraint family). They have been implemented by taking advantage of the underlying constraint solvers in SWI-Prolog. We have added types to programs so that relations become typed (as tables in relational databases) and each constraint is mapped to its solver.

The big difficulties in the implementation of our stratified fixpoint semantics consist of the adaptation of the usual techniques for not only working with constraints but also taking into account that a database can dynamically be augmented with local clauses, when an hypothetical query is formulated. The definition of the fixpoint operator is not constructive for the case of nested implications, then a stronger definition of dependency graph has been formulated to ensure a constructive and terminating fixpoint computation.

Future work The prototype presented in this work can be enhanced to set it as a practical system. The current implementation is very close to the theory developed in our previous works and is a valuable tool for understanding such a theory, but as a consequence it has an expected penalty in efficiency. On the one hand, we have implemented a naïve stratification algorithm for this first prototype that can be easily improved. On the other hand, a more serious source of inefficiency comes from the forcing of implication. In this line, well-known methods as magic set transformations [Beeri and Ramakrishnan 1991] and tabling [Tamaki and Sato 1986] could be worth to be adapted to the current implementation. This is also related to widen the set of computable queries and programs, by adapting the ideas found in the well-founded model [Van Gelder et al. 1991], that could relax our stratification restrictions. This can also be coupled with efficient solving methods [Shen et al. 2002]. In addition, to use existing efficient relational technology to solve concrete queries which do not need the more powerful (less-efficient) database engine we currently provide.

Moreover, in the field of databases, the useful constraint systems are often combinations of different domains. The constraint systems we have implemented work together, but do not cooperate. Due to the nature of the logic involved in our system, finding methods for proving satisfiability of constraints in a mixed domain is a complex task, because the syntax of such constraints will allow, among other aspects, combining existential and universal quantifications for variables of the considered domains. In order to develop a mixed solver, we will consider the existing works that combine concrete domains in the context of $HH_\neg(\mathcal{C})$ [García-Díaz and Nieva 2003] and the combination of decision methods with techniques applied to constraint solvers. This line comes from a fruitful research line in combining constraint systems to cope with problems that, either cannot be handled by a domain constraint solver alone, or its solving can be significatively improved by cooperation of constraint solvers [Hofstedt and Pepper 2007, Castro and Monfroy 2004, Granvilliers et al. 2001].

Acknowledgments

This work has been partially supported by the projects STAMP (TIN2008-06622-C03-0)1, PROMESAS-CAM (S-0505/TIC/0407) and UCM-BSCH-GR58/08-910502. We are also grateful to Jan Wielemaker, author of SWI-Prolog, and Markus Triska, author of the finite domain constraint library for this system, who was very kind to support us in developing new features used in our finite domain constraint solver.

References

F. Arni, K. Ong, S. Tsur, Haixun Wang, and C. Zaniolo. The Deductive Database System LDL++. *Theory and Practice of Logic Programming*, 3(1):61–94, 2003.

M. Becker, C. Fournet, and A. Gordon. Design and Semantics of a Decentralized Authorization Language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, WashingtonFrancesco, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2819-8. doi: `http://dx.doi.org/10.1109/CSF.2007.18`.

C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3-4):255–299, 1991. ISSN 0743-1066. doi: http://dx.doi.org/10.1016/0743-1066(91)90038-Q.

A. Calì, G Gottlob, and T. Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 14–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-423-2. doi: http://doi.acm.org/10.1145/1514894.1514897.

C. Castro and E. Monfroy. Designing hybrid cooperations with a component language for solving optimisation problems. In *International Conference on Artificial Intelligence: Methodology, Systems and Applications 2004*, volume 3192 of *LNCS*, pages 447–458. Springer, 2004.

R. Fikes, P. J. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. *Journal of Web Semantics*, 2(1):19–29, 2004. URL http://www.informatik.uni-trier.de/~ley/db/journals/ws/ws2.html#FikesHH04.

M. García-Díaz and S. Nieva. Solving Constraints for an Instance of an Extended CLP Language over a Domain based on Real Numbers and Herbrand Terms. *Journal of Functional and Logic Programming*, 2003 (2), September 2003.

L. Granvilliers, E. Monfroy, and F. Benhamou. Cooperative solvers in constraint programming: a short introduction. *ALP Newsletter*, 14(2), 2001.

P. Hofstedt and P. Pepper. Integration of declarative and constraint programming. *Theory and Practice of Logic Programming*, 7(1-2):93–121, 2007. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068406002833.

C. Holzbaur. Realization of forward checking in logic programming through extended unification. Report TR-90-11, Oesterreichisches Forschungsinstitut fuer. *Artificial Intelligence*, 1990.

M. Jarke, M. A. Jeusfeld, and C. Quix. ConceptBase V7.1 User Manual. Technical report, RWTH Aachen, April 2008.

M. S. Lam, S. Whaley, B. V. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In Chen Li, editor, *Symposium on Principles of Database Systems (PODS)*, pages 1–12. ACM, 2005. ISBN 1-59593-062-0.

J. Leach, S. Nieva, and M. Rodríguez-Artalejo. Constraint Logic Programming with Hereditary Harrop Formulas. *Theory and Practice of Logic Programming*, 1(4):409–445, 2001.

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Towards a constraint deductive database language based on hereditary harrop formulas. In P. Lucio and F. Orejas, editors, *Sextas Jornadas de Programación y Lenguajes, PROLE*, pages 171–182, 2006.

S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08, Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 289–304, Ise, Japan, 2008. Springer-Verlag.

G. Ramalingam and Eelco Visser, editors. *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, 2007. ACM. ISBN 978-1-59593-620-2.

R. Ronen and O. Shmueli. Evaluating very large datalog queries on social networks. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 577–587, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-422-5. doi: http://doi.acm.org/10.1145/1516360.1516427.

F. Sáenz-Pérez. Datalog Educational System. User's Manual version 1.6.2. Technical report, Faculty of Computer Science, UCM, march 2009. Available from http://des.sourceforge.net/.

K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM. ISBN 0-89791-639-5. doi: http://doi.acm.org/10.1145/191839.191927.

Y. Shen, L. Yuan, and J. You. Slt-resolution for the well-founded semantics. *Journal of Automated Reasoning*, 28:53–97, 2002.

H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings on Third international conference on logic programming*, pages 84–98, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-16492-8.

A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/116825.116838.

J. Wielemaker. SWI-Prolog. User's Manual version 5.6.64, 2009. Available from http://www.swi-prolog.org/.

C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Pages 180–183, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-443-X.

Incorporating Integrity Constraints to a Deductive Database System

Gabriel Aranda-López[†], Susana Nieva[†], Fernando Sáenz-Pérez[‡] and Jaime Sánchez-Hernández[†]

[†] *Dept. Sistemas Informáticos y Computación and* [‡] *Dept. Ingeniería del Software e Inteligencia Artificial,*
Universidad Complutense de Madrid, Spain
{nieva,fernan,jaime}@sip.ucm.es, garanda@fdi.ucm.es

Abstract

Hereditary Harrop Formulas with Constraint and Negation ($HH_{\neg}(\mathcal{C})$) have been proposed as a very expressive constraint deductive database scheme. The theoretical foundations lay on a fixpoint semantics that is also the basis for a Prolog implementation presented in a previous work. We have developed several solvers for specific constraint domains. In this paper we introduce, for the first time, (strong) integrity constraints in the $HH_{\neg}(\mathcal{C})$ system by taking advantage of the expressiveness of our approach. Integrity constraints are used to ensure consistency of data in a database language. A (strong) integrity constraint expresses a relationship among data that every database instance is required to satisfy. We show how our language and the fixpoint semantics implementation lead to an easy specification of the most usual integrity constraints provided in other relational database languages. In addition to the usual specifications of *primary key* and *foreign key* the system also supports *functional dependencies*.

Keywords: Deductive Databases, Constraints, Hereditary Harrop Formulas, Fixpoint Semantics, Integrity Constraints

1 Introduction

In [7] we presented an extension of Hereditary Harrop formulas with constraints by adding negation to obtain $HH_{\neg}(\mathcal{C})$, a Constraint Deductive Database (CDDB) system as Datalog (with Constraitnts) [9], based on a fixpoint semantics as Coral [8]. We stress, as an important benefit of our approach, the ability to formulate hypothetical and universally quantified queries. The resulting language enjoys the expressive power of Datalog, but adds constraints and two new logical connectives: implication (to formulate hypothetical queries), and universal quantification (to encapsulate data). We

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

have implemented a prototype as proof of concept for the theoretical framework. Two main components can be distinguished in the implementation of this prototype, that we already presented in [1]. One corresponds to the bottom-up implementation of the fixpoint semantics, which is very close to the theory. The fixpoint is computed using a stratification of the predicates of the database which is obtained from a suitable notion of dependency graph. The other component corresponds to the implementation of the constraint solvers. The first component is independent of the particular constraint system, i.e., it is parametric on the second component. Then, the known safety results for Datalog (with constraints) [9] are valid in our case, because they rely on the constraint systems.

In [3] we shown a first approximation of how incorporate aggregate functions to our system. Those functions are useful in computing single values from a set of numerical or other-type values. We have taken advantage of certain aspects of the stratified semantics of our database system in order to deal with the implementation of aggregate functions.

$HH_{\neg}(\mathcal{C})$ prototype incorporates a type checking and inferrer system, then our clauses defining databases are typed. Types are needed to know the constraint system the constraints belong to. We have proposed three constraint systems as possible instances of the scheme $HH_{\neg}(\mathcal{C})$: Boolean, Reals, and Finite Domains.

We have relied on the underlying constraint solvers available in SWI-Prolog [12] for implementing the constraint systems. In addition, due to some conectives of the language are not suppoted in SWI-Prolog (for instance universal quantifier), it has been necessary to make an explicit management for them.

Consistency constraints over data are known as strong integrity constraints in the deductive database area. Examples of such integrity constraints in the relational model are primary keys and foreign keys, to name a few. Such integrity constraints must not be missed with those belonging to constraint system $\mathcal{C}$ used to parameterize the scheme $HH_{\neg}(\mathcal{C})$ (or analogously $CLP(\mathcal{C})$, where constraints are first-class citizen constructions of the language, as they can occur in well-formed formulas. As well, constraints in deductive systems have also been studied [2,5], as DLV [6] or XSB [10] implementing stable model [4] and well-founded model semantics [11], respectively, are otherwise understood as model filters. Since a database can have several models, only those complaining constraints are included in the answer, therefore discarding *unfeasible* models from the answer.

In this paper, instead, we focus on integrity constraints as understood in relational database management systems (RDBMS's) in order to provide a means to detect inconsistent data with respect to user requirements. Our system already included one such kind of a constraint in the form of *domain* constraints (as known is RDBDM's) as it is strongly typed. Here, we describe

for the first time the introduction of other integrity constraints: the more usual primary key and foreign key, and also the functional dependency, which in particular is useful for ensuring consistency over denormalized relations. We present how integrity constraints can be specified at the language level using $HH_{\neg}(\mathcal{C})$ expresiveness. We also explain the issues which arise in enabling integrity constraints support in a general database framework. We propose a concrete implementation and we sketch some improvements to enhance performance.

2 System Description and Fixpoint by Strata

In this section we introduce the main aspects of our system and how to use it.

2.1 Databases and Queries

As usual in CDDBs, a database is a set of clauses and a query is a goal, whose answer is a constraint. When the system is started, the computation of the fixpoint semantics of a database Δ follows the next computation stages:

(i) Check and infer the predicate types.

(ii) Build the dependency graph of Δ.

(iii) Compute a stratification s for Δ, if there is any. Otherwise, the system throws an error message and stops.

(iv) If the previous step succeeds, compute the fixpoint of Δ, $fix(\Delta)$, from the first to the last stratum.

The fixpoint $fix(\Delta)$ is compossed by a set of pairs (A, C) such that the atom A can be derived from Δ if C is satisfied, i.e., it captures the semantics of the database Δ. Those pairs are obtained by means of a stratified fixpoint operator, whose computation is specially difficult when dealing with nested implications. This is because when an implication $D \Rightarrow G$ appears in the body of a clause, the database for which the fixpoint is being computed is augmented with D. Then a local fixpoint for the extended database must be calculated. See [1] for details.

After the stage (iv), the system keeps in memory, while not processing another database, the following information: the just computed $fix(\Delta)$, the stratification s, and the dependency graph of Δ

When a query `G` is posed at the prompt, the system computes, if it exists, a new stratification s' for the set $\Delta \cup \{\texttt{G}\}$. The query can not be computed if there is not such s', and the systems stops. In other case:

- If $s = s'$, the kept fixpoint, computed for Δ, is valid to evaluate `G`.
- If $s \neq s'$, the symbol predicates involved in the computation of `G` can be in a

different stratum than when $fix(\Delta)$ was computed. So, the stored fixpoint is not valid now to evaluate `G` and a new fixpoint, $fix(\Delta \cup \{\texttt{query}(\overline{\texttt{X}})\texttt{: -G}\})$, must be computed, where $\overline{\texttt{X}}$ are the free variables of `G`. The new predicate `query` added to the database captures the desired answer, that will be the constraint `C` stored in the pair `(query(`$\overline{\texttt{X}}$`),C)` of the computed fixpoint. More details of the implementation are shown in [1]. In the next section we introduce the use of $HH_{\neg}(\mathcal{C})$ system by means of examples.

2.2 A Database Example

Here we define a simple database for managing information about students in some courses related to *Logic Programming*. We adhere to a syntax quite similar to Prolog. In addition, we write `not` for negation, `=>` for implication, `ex(X,G)` representing $\exists$`X G`, and `fa(X,G)` representing $\forall$`X G`. First, we define two domains, one for the student names and the other for subject names:

```
domain(student_dt,[angela,david,joseluis,nicolas]).
domain(subject_dt,[programming_introduction,logic_programming,
                   declarative_programming]).
```

Although several solvers can be used together within the same database, they can not be combined for the moment, i.e., constraints of different types cannot be freely mixed to get an heterogeneous compound constraint. Predicates with arguments of different types are restricted to those extensionally defined.

This limitation can be surpassed in some practical situations by defining domain mappings. In this example, we can easily define a mapping from the domain of student names to real values, i.e., associate a real number (intended as the identity card number) to each student:

```
type(student_id(student_dt,real)).
student_id(angela,350001).
student_id(david,500002).
student_id(joseluis,750003).
student_id(nicolas,900004).
```

We also associate each subject name to its code:

```
type(subject_id(subject_dt,real)).
subject_id(programming_introduction,406).
subject_id(logic_programming,428).
subject_id(declarative_programming,455).
```

The information about students taking a course can be stored as:

```
type(course(real,real,real)).
```

```
course(350001, 406, 5.0).
course(750003, 406, 7.5).
course(500002, 406, 2.0).
course(350001, 428, 3.5).
```

For instance, the first clause means that the student with identity card number `350001` is taking the course of *Programming Introduction* (identifier `406`) and it has a mark of `5.0`.

We focus now on the intensional database. In a first view, we collect the students `St` who have passed a concrete subject `Sb`:

```
type(passed(real,real)).
passed(St,Sb) :- course(St,Sb,M), constr(real ,M>=5.0).
```

The next view represents that a student is able to register in *Declarative Programming* (identifier `455`) if it has passed *Programming Introduction* (identifier `406`) and it is already registered in *Logic Programming* (identifier `428`):

```
type(register(real,real)).
register(St,455) :- passed(St,406), course(St,428,M).
```

Fixpoint of the Database

When the user process (load) a database, the system performs the computation stages mentioned in Section 2.1, and obtains a fixpoint for such a database. That fixpoint is a set of pairs `(Atom,Constraint)`, meaning that the atom is true if the constraint is satisfied. The fixpoint for our working example is:

```
passed(A_real, B_real), (B_real=406.0, A_real=350001.0;
                         B_real=406.0, A_real=750003.0)
register(350001.0, 455.0), true
student_id(angela, 350001.0), true
student_id(david, 500002.0), true
student_id(joseluis, 750003.0), true
student_id(nicolas, 900004.0), true
subject_id(programming_introduction, 406.0), true
subject_id(logic_programming, 428.0), true
subject_id(declarative_programming, 455.0), true
course(350001.0, 406.0, 5.0), true
course(750003.0, 406.0, 7.5), true
course(500002.0, 406.0, 2.0), true
course(350001.0, 428.0, 3.5), true
```

Querying

Once the database is loaded the user can submit queries to the system. For example, the user can ask how many students are studing *Programming Introduction* (identifier `406`):

```
HHn(C)> constr(real,NumSt=count(course(St,406.0,M))).
Answer: NumSt=3.0
```

Who is not able to register in *Declarative Programming* (identifier `455`):

```
HHn(C)> not(register(St,455.0)),student_id(N,St).
Answer: ( N=nicolas, St=900004.0;
          N=joseluis, St=500003.0;
          N=david, St=750002.0;
          N=angela, St=350001.0),
        St/=350001.0
```

Assuming that a student has passed *Programming Introduction* (identifier `406`) with a mark of `9.0`, what is the average mark of the class in this subject:

```
HHn(C)> course(750003.0,406.0,9.0)=>
        constr(real,Avg=avg(course(St,406.0,C),C)).
Answer: Avg=5.875
```

The current version of the system is available at https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems including a bundle of examples.

3 Integrity Constraints in $HH_{\neg}(\mathcal{C})$

In this section we introduce integrity constraints in the system. We describe the different approaches that we have considered dealing with this feature. Firstly, we show how the language by itself is able to capture the most common integrity constraints, at least for databases that do not involve some forms of nested implications (in particular, the specifications will be sound for the extensional database, which is the most common use in relational databases). Then, we explain the problems coming from nested implications and the form in which they can be overcome within the concrete implementation of the system.

3.1 Using Expressiveness of the Language to Define Integrity Constraints

The intended aim when designing the $HH_{\neg}(\mathcal{C})$ language is to get a highly expressive query language. As a proof of concept, in this section we show how the user can define (strong) integrity constraint within the laguage itself, by adding predicates to the input database that capture information about the violation of these constraints. For the examples below, we suppose an

alphabetical domain `data` composed of all letters of the alphabet. Let us start with a *primary key* constraint for a defined predicate `p`:

```
p(a,b,c).
p(b,c,d).
p(e,f,g).
p(a,i,j).
```

A *primary key* constraint specifies that there are no two tuples in a predicate with the same values for a given set of columns. For example, for specifing that the first parameter is the primary key for the predicate `p`, we add to the database:

```
pk_p_fails:- p(A,B,C), p(A,B2,C2),constr(data,(B/=B2;C/=C2)).
```

The predicate `pk_p_fails` express a failure in the intended primaray key constraint, i.e., it is true iff the constraint is violated. In this case `pk_p_fails` is true as the value `a` occurs twice as the first argument of `p`. Moreover, we can capture information about the tuples that violate the constraint by adding parameters in the head:

```
pk_p_fails(A,B,C):- p(A,B,C), p(A,B2,C2),
                    constr(data,(B/=B2;C/=C2)).
```

In this case, those tuples are `(a,b,c)` and `(a,i,j)`.

A *foreign key* constraint specifies that the values in a given set of columns of a predicate must exist already in the columns declared in the primary key constraint of another predicate. Assume for example the following predicates `s` and `q`:

```
s(a,b).     q(a,b).
s(e,a).     q(e,f).
s(h,e).     q(h,j).
s(k,k).     q(k,l).
s(a,b).     q(c,d).
s(a,c).     q(t,g).
```

For defining a foreign key between the first argument of `s` and the first argument of `q` we are interested in those tuples that could be defined as:

```
fk_not_pairs(X):-q(X,A),not(s(X,B)).
```

With this idea it is straightforward to define the `fk_qs_fails` predicate:

```
fk_qs_fails(V):- q(V,A), fa(B,not_s(V,B)).
not_s(V,B):- not(s(V,B)).
```

In this example, this predicate will capture the concrete values `(V=c; V=t)`.

A *functional dependency* constraint $\overline{X} \rightarrow \overline{Y}$ over a predicate p specifies that the set of arguments $\overline{X}$ of p functionally determine the set $\overline{Y}$, i.e., each

tuple of values of $\overline{X}$ in p is associated with exactly one tuple of values $\overline{Y}$ in the same tuple of p. For instance assume a predicate `u`:

```
u(a,b,c,d,e,f).
u(a,b,d,c,e,f).
u(a,a,a,a,a,a).
u(a,b,c,h,a,a).
```

For the predicate `u(A1,A2,B1,B2,C,D)`, let us assume the functional dependency $(\texttt{A1}, \texttt{A2}) \rightarrow (\texttt{B1}, \texttt{B2})$. Its violation can be expressed by the following predicate:

```
fd_u_fails(A1,A2,B1,B2):- u(A1,A2,B1,B2,_,_),
                          u(A1,A2,B3,B4,_,_),
                          constr(data,(B1/=B3 ; B2/=B4)).
```

This predicate `fd_u_fails(A,B,C,D)` will capture the tuples:

```
(A=a, C=c, D=d, B=b;
 A=a, C=c, D=h, B=b;
 A=a, C=d, D=c, B=b).
```

In the previous examples the user realizes that a violation of an integrity constraint occurs because of the presence of pairs for concrete predicates in the database fixpoint. This is a direct first approach, that shows the expressiveness of the language and also serves as a clear specification of the integrity constraints we are interested in. Nevertheless, as we have pointed out before this specification is not complete for the general case. Nested implications require a sophisticated computation mechanism involving temporary fixpoint calculations (see [1] for a detailed description of this mechanism). The information about integrity constraint violation, as previously defined, may appear in some temporary fixpoint but not in the final fixpoint. To illustrate this situation, assume the following database:

```
q(a,b).
p:- q(a,c) => q(a,c).
```

And a definition of primary key for the first argument of `q`, following the previous ideas:

```
pk_q_fails:- q(A,B), q(A,C), constr(data,B\=C).
```

During the computation for the predicate `p` it is required to calculate a local fixpoint for an extended database including the tuple `q(a,c)`. At this point, both tuples `q(a,b)` and `q(a,c)` are in the fixpoint, which means a violation of the specified primary key. In fact, the pair `(pk_q_fails,true)` will be added to this local fixpoint, that is not kept anymore. So the pair `(pk_q_fails,true)` does not appear in the final fixpoint, which is:

```
p, true
q(a, b), true
```

And this is not consistent according to the definition of primary key for `q`. It is interesting that at some point of the computation the inconsistency is present and could be checked by the system with some easy modifications of it. In the next section we focus on these issues in order to get a practical implementation.

3.2 Adding Integrity Constraint Support to a Database System

Three issues must be taken into account for adding support to integrity constraints to a given database system and language.

First, *how* integrity constraints are declared. On the one hand, RDBMS's allow to declare them with both DDL (Data Definition Language) and GUI interfaces (in turn, the latter uses the former and hidden from the user). On the other hand, Prolog systems include either assertions or directives for such declarations. Currently, although somewhat interactive, our system reads a database which is not subject to change up to next loading, i.e., it is more targeted to be used as a batch database instead of a full fledged online database. So, including a DDL in addition to the current DML (Data Manipulation Language) does not seem a need. Thus, as $HH_{\neg}(\mathcal{C})$ is a logic programming (LP) language, we rather follow the more usual way in these LP systems and provide support to assertions. Next section presents the concrete syntax we propose.

Second, *when* a constraint must be detected as violated. Our system implements an iterative fixpoint procedure with a common operation for adding new tuples to the current interpretation. So, a safe check to detect constraint violation is at this point. This can be seen as a sufficient condition, but weaker conditions can be found, indeed. Given that each constraint is implemented with a predicate, each time a given atom and constraint is to be added, its predicate name can be checked for matching with those constraint predicates. Other alternatives do exist, from this just introduced end to the other, opposite end, when constraints are checked after all fixpoint computation is finished. But notice that, as implications are allowed, it might be the case that some constraints had been violated in a given subcomputation that does not longer be included in the *outcome* fixpoint, as explained in the previous section.

And, third, *what* to do when an inconsistency is found. Here, batch and online operations are considered as a reference to this question. Online updates to databases usually do constraint checking for each tuple to be either added, modified or deleted, discarding the update if an inconsistency is found and raising an exception. Batch updates, on the other hand, can follow another route: Collect the inconsistencies and report offending tuples to the user. This

is quite adequate for massive updates for which it is convenient to have all the offending tuples at hand in order to reason about and repair error sources. Several RDBMS's do allow these batch updates with different tools (Oracle, DB2, MySQL, ...).

3.3 Concrete Implementation: The Case of Primary Key

In this section we explore a first approach for a concrete implementation of integrity contraints, that is currently working in the system. The aim is to automate the generation of the specifications introduced in Section 3.1, taking reasonable decisions about the three questions presented in the previous Section. The problem explained in Section 3.1 about nested implications and the associated subcomputations can be surpased now at the system implementation level.

We focus on primary key constraints, as the others (foreign keys, and functional dependencies) can be treated in a similar way. For the first question *how*, we introduce a new syntactical contruction in the system in order to define a primary key contraint. Given a predicate p of arity n, and two tuples of variables $\overline{X}$ and $\overline{Y}$ (of arities n and $m \leq n$ resp.) such that $\overline{Y} \subseteq \overline{X}$, then the directive $pk(p(\overline{X}), \overline{Y})$ establishes the arguments corresponding to $\overline{Y}$ as a primary key for p. For example, the primary key constraint for the predicate `p` of Section 3.1 would be expressed as:

```
:- pk(p(A,B,C),(A))
```

The system automates the translation of this directive, producing exactly the code for `pk_p_fails(A,B,C)` presented in Section 3.1.

The system must also check the possible violations of integrity contraints. As explained in Section 3.1, we can not wait for the complete fixpoint of the database. But the system implements an iterative fixpoint procedure with a specific operation for adding new pairs to the current interpretation. Then, for the second question, *when*, a safe answer is whenever a new pair is added to the current interpretation, as any fixpoint (temporary or not) comes from a growing interpretation. Of course, this solves the problem pointed out in Section 3.1 related to temporary fixpoints. Note that this is analogous to the approach taken on by current RDBMS implementations. The idea is to avoid *speculative* computations by detecting inconsistencies as soon as possible. In our case, there is another natural point to check integrity that could also be implemented, that is to check it at the end of any fixpoint computation.

For the last question, *what*, we have decided to show the error message with the corresponding predicate and the values of the tuples that cause the conflict.

For instance, for the primary key constraint violation of predicate `p`, of Section 3.1, this message is:

```
-- Integrity Constraint Violation --
Duplicate primary key in predicate p, duplicate value: a
```

After this error message, the system continues and finishes the computation.

4 The next steps

Following the line of the first approach of Section 3.1 we plan to use $HH_{\neg}(\mathcal{C})$ expresiveness to declare user-defined integrity constraints as views. At a first easy example, referred to the student database in Section 2, we can define an user integrity constraint aimed to limit the number of students of a course:

```
lp_quote_exceeded :- constr(real,count(course(X,406,M))>25).
```

So, an integrity constraint specifies *unfeasible* values rather than feasible. This is opposed to the usual way of specifying integrity constraints in SQL with `CHECK` clauses, where the positive case is specified instead of the negative one as we do. Notice also that in this example we take advantage of our constraint system implementation, which supports aggregates. Following the previous philosophy, it is straightforward to define a system directive for user-defined constraints. For this example:

```
:- uc(lp_quote_exceeded).
```

This directive would specify that the predicate `lp_quote_exceeded` should not occur in any fixpoint. Otherwise, an informative error message would be raised.

In addition, we plan to explore alternative implementations of the integrity constraints. An alternative is to keep all the information about integrity constraints separated from the fixpoint, i.e., integrity constraints does not affect to the semantics of the database. They are interpreted as *external observers* of that fixpoint that would raise information about the semantic inconsistencies defined by integrity constraints.

Acknowledgements: This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502).

References

[1] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas. In *Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09)*, pages 117–128. ACM Press, 2009.

[2] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 14–30, New York, NY, USA, 2009. ACM.

[3] G.Aranda, S.Nieva, F.Sáenz-Pérez, , and J. Sánchez. A prototype constraint deductive database system based on $hh_{\neg}(\lfloor)$. In *X Jornadas sobre Programación y Lenguajes (PROLE'10)*, pages 189–196, 2010.

[4] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[5] Robert Kowalski, Fariba Sadri, and Paul Soper. Integrity checking in deductive databases. In *In Proceedings of the VLDB International Conference*, pages 61–69. Morgan Kaufmann Publishers, 1987.

[6] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[7] S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08, Proceedings*, volume 4989 of *LNCS*, pages 289–304, Ise, Japan, 2008. Springer-Verlag.

[8] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL Deductive System. *The VLDB Journal*, 3:161–210, 1994.

[9] P. Z. Revesz. *Introduction to Constraint Databases*. Springer, 2002.

[10] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM.

[11] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

[12] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

The Journal of Logic and Algebraic Programming 83 (2014) 20–52

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

www.elsevier.com/locate/jlap

An extended constraint deductive database: Theory and implementation

Gabriel Aranda-López *, Susana Nieva, Fernando Sáenz-Pérez, Jaime Sánchez-Hernández

Facultad de Informática, Complutense University of Madrid, Spain

ARTICLE INFO

Article history:
Received 28 July 2011
Received in revised form 4 April 2013
Accepted 3 July 2013
Available online 9 July 2013

Keywords:
Deductive databases
Constraints
Hereditary Harrop formulas
Fixpoint semantics

ABSTRACT

The scheme of Hereditary Harrop formulas with constraints, $HH(\mathcal{C})$, has been proposed as a basis for constraint logic programming languages. In the same way that Datalog emerges from logic programming as a deductive database language, such formulas can support a very expressive framework for constraint deductive databases, allowing hypothetical queries and universal quantifications. As negation is needed in the database field, $HH(\mathcal{C})$ is extended with negation to get $HH_{\neg}(\mathcal{C})$. This work presents the theoretical foundations of $HH_{\neg}(\mathcal{C})$ and an implementation that shows the viability and expressive power of the proposal. Moreover, the language is designed in a flexible way in order to support different constraint domains. The implementation includes several domain instances, and it also supports aggregates as usual in database languages. The formal semantics of the language is defined by a proof-theoretic calculus, and for the operational mechanism we use a stratified fixpoint semantics, which is proved to be sound and complete w.r.t. the former. Hypothetical queries and aggregates require a more involved stratification than the common one used in Datalog. The resulting fixpoint semantics constitutes a suitable foundation for the system implementation.

1. Introduction

The extension of *LP* (Logic Programming) with constraints gave rise to the *CLP* (Constraint Logic Programming) scheme [26,25]. In a similar way, the $HH(\mathcal{C})$ scheme (Hereditary Harrop formulas with Constraints) [31,19] extends *HH* by adding constraints. In both cases, a parametric domain of constraints is assumed for which it is possible to consider different instances (such as arithmetical constraints over real numbers or finite domain constraints). The extension is completely integrated into the language: constraints are allowed to occur in goals, bodies of clauses, and answers.

As a programming language, $HH(\mathcal{C})$ can still be viewed as an extension of *CLP* in two main aspects. On the one hand, the logic *HH* introduces new connectives which are not available in Horn Clause logic, such as disjunction, implication and universal quantifiers [36]. On the other hand, and following Saraswat [45], in the scheme $HH(\mathcal{C})$, the notion of constraint system is established in such a way that any $\mathcal{C}$ satisfying certain minimal conditions can be considered as a possible instance for the scheme. In [45], as minimal conditions the language of constraints incorporates $\wedge$ and $\exists$. However, particular constraint systems may include more logical symbols as $\forall$ and $\Rightarrow$, together with the corresponding assumptions related to their behavior. Therefore, the language of constraints itself extends the common ones used in *CLP*, consequently facilitating the representation of more complex constraints.

* Corresponding author.
E-mail address: garanda@fdi.ucm.es (G. Aranda-López).

1567-8326/$ – see front matter
http://dx.doi.org/10.1016/j.jlap.2013.07.002

This paper extends other works [38,2] in which we investigated the use of $HH(\mathcal{C})$ not as a (general purpose) programming language, but as the basis for constraint deductive database (*CDDB*) systems [30,42]. The motivation is that, in the same way that Datalog [51,56] and Datalog with constraints [27] arise for modeling database systems inspired by *LP* and *CLP* respectively, the language $HH(\mathcal{C})$ can offer a suitable starting point for the same purpose. We show that the expressive power of $HH(\mathcal{C})$ improves existing languages by enriching the mechanisms for database definition and querying, with new elements that are useful and natural in practice. In particular, implications can be used to write hypothetical queries, and universal quantification allows encapsulation. The existence of constraints is exploited to represent answers and to finitely model infinite databases and answers. This is also the case of constraint databases, but the syntax of our constraints is also more expressive than the one commonly used in them, as it is the case of Datalog with constraints.

However, $HH(\mathcal{C})$, as it was originally introduced, lacks negation which, as we will see, is needed for our proposal to be complete with respect to relational algebra (*RA*). We have extended $HH(\mathcal{C})$ with negation, to obtain $HH_{\neg}(\mathcal{C})$ to be used as a database language. We have defined a proof-theoretic semantics to provide the meaning of goals (queries) and programs (databases). This meaning is represented by answer constraints, which can be obtained using the goal-oriented rules of a sequent calculus which combines intuitionistic inference rules with deductibility in a generic constraint system. Also, a stratified fixpoint semantics has been defined and proved to be sound and complete with respect to the previous one. The motivation for introducing this new semantics take into account several aspects:

- The fixpoint semantics provides a model for the whole database, while the proof-theoretic one (as well as top-down semantics in general) focuses only on the meaning of a query in the context of a database. In fact, the fixpoint of a database will correspond to the instance of the database. Thereby, the fixpoint semantics supplies a framework in which properties such as equivalence of databases can be easily analyzed, which gives formal support to the study of query optimization.
- In order to deal with recursion and negation, we have followed the stratified negation approach used in [51] which gives semantics to Datalog. The use of a fixpoint semantics as an operational mechanism has been adopted as a good choice in several deductive database systems as it is able to avoid the non-monotonicity inherent to negation. Then it guarantees termination, as long as the constraint answer sets are finite. Further, it facilitates to work with different constraint systems, relegating the problem of termination to the problem of finding intensional representations of data by the constraint system. This issue is dealt in works as [40], where safety conditions are imposed to the constraint system, and some particular systems are identified as satisfying such conditions. Hence, termination for any query is ensured for these systems. But, identifying such particular systems is out of the scope of this work.
- Introducing negation in goals makes a given database to may have several meanings [51]. Stratified negation is one of the approaches that, by imposing syntactical restrictions, guarantees a unique model for the database: the minimal fixpoint interpretation. Moreover, stratification has been previously used as a useful resource when dealing with hypothetical queries in Datalog [9], in order to get a unique minimal model for the database, as it is the case of our proposal.
- Stratification is a common technique to deal with aggregates [42], because it ensures monotonicity. Our stratified design of the fixpoint semantics has become a good framework to implement aggregates.

In order to define a stratified fixpoint semantics for $HH_{\neg}(\mathcal{C})$, we have adapted the usual notion of dependency graphs to include the dependencies derived from implications inside goals, as well as those derived from aggregate functions. The fixpoint of a database is computed as a set of pairs (A, C), where A is an atom and C a constraint. The atom A can be understood as an n-ary relation instance, where its arguments are constrained by C. According to the dependency graph, predicates are classified by strata and these pairs are computed by strata. Each stratum should become saturated before trying to saturate any other higher stratum. However, as an implication may occur in a goal, the computation must take into account that the database is augmented with the hypothesis posed in the implication antecedent. From the theoretical point of view, this issue does not make any obstacle, but it can be a drawback for a concrete implementation. In our approach, the fixpoint of the augmented database must be locally computed to solve the implication. But our proposal takes advantage of the stratification to avoid cycles during the computation. In addition, the dependency graph can be useful to reduce this computation to the part of the database that is involved in the implication.

The fixpoint semantics provides support for a concrete database system. We have implemented a Prolog prototype very close to the underlying theory as a proof-of-concept. Essentially, it incorporates all the features introduced in the paper, and moreover it supports aggregate functions. Two main components can be distinguished in the implementation of this database language. One corresponds to the implementation of the fixpoint semantics which is independent of the concrete constraint system. The other component corresponds to the implementation of the constraint system. We have considered and implemented solvers for the following constraint systems: Boolean, Reals, and Finite Domains, as instances of $\mathcal{C}$ in the scheme $HH_{\neg}(\mathcal{C})$. The implementation is designed in such a way that more than one constraint system can be used within the same database. We have designed a type system for identifying the constraint system each constraint in a database belongs to.

1.1. Related work

It is well known that negation in logic programming is a difficult issue and there has been a great amount of work about it [1], and also in constraint logic programming [46] and deductive databases [8] since long time ago. A first bag of issues

comes from deriving incorrect answers or failing to derive others in presence of classical negation. Several problems arise in this setting, as unsoundness of *SLDNF*, which is workarounded by restricting a negative goal to be selected until it becomes ground [34]. However, this introduces another problem: floundering [5], which is avoided in the field of deductive databases with safety conditions [51]. Safety conditions have been also applied to constraint deductive databases [7,40,41] for particular constraint systems, enabling to develop closed-form constraints as answers. The closed-form evaluation requirement guarantees that it is possible for the query solver to calculate intensional forms for the answer. In constraint databases, where (non-ground) intensional data are managed, constructive negation [29,14,16] can be used instead of such safety conditions. Its rationale lies on allowing non-ground negative goals, which can construct answers by involving constraints on goal variables, therefore avoiding floundering. This is the very fundamental of constraint databases [42], where an answer to a negated goal is a set of constraints. This idea was used in *CLP* [46] as an approach to constructive negation to avoid floundering. Our work can also be seen from this perspective because the answer to a negated goal is also constructive. Nevertheless, the proposal of [46] is based on classical logic, while our approach is based on intuitionistic logic including implication and universal quantification.

A second bag of issues comes from assigning a model to a program valid to the user under an intended semantics. As mentioned before, stratified negation guarantees that only one minimal model can be assigned to a program [51]. Other approaches are based on non-monotonic logic, as inflationary semantics [13], which is also based on a two-valued logic and assigning one model to a program. Its drawback is that, in general, that model is not a minimal one and inflationary model semantics does not always meet the intended semantics. Next approaches are based on three-valued logic and produce in general several outcome models:

Gelfond and Lifschitz proposed Stable Models [21], a declarative semantics for logic programs with negation, based on autoepistemic logic. Another related approach is the Well-Founded semantics defined by Van Gelder et al. [52], where the main idea is the notion of *unfounded set*, which provides the basis for obtaining negative information in the semantics. Answer Set Programming (*ASP*) is a form of declarative programming oriented towards difficult combinatorial search problems [32,20]. It is based on the work about Stable Models and fast propositional solvers are used as computational mechanism for inference. Its key idea is to use ground instances of programs. Our scheme $HH_{\neg}(\mathcal{C})$ is designed to work with any generic constraint language $\mathcal{L}_{\mathcal{C}}$ and the use of ground instances would impose a serious limitation on the constraint systems allowed as instances of the scheme. This technique would be adequate for a Herbrand constraint system, but they are unfeasible for more sophisticated constraints. Notice that $HH_{\neg}(\mathcal{C})$ constraint systems work with intensional representations that are able to be more general than a simple equality. For example, constraints over real numbers should be excluded as a possible instance, as no grounding is possible (at least in a straightforward way) and it would be a serious drawback for our proposal. However, although *ASP* includes constraints, they are viewed as integrity constraints which discards models in the answer rather than syntax objects which can be dealt as first citizen constructions *á la CLP*. In addition, neither implications nor quantifiers are supported. It might be expected that introducing the implication, which dynamically produces an increasing of the database, in a system based on the approaches just mentioned, additional computations would be necessary, as it happens in our system.

Although stratification imposes certain syntactic restrictions to the language, these conditions are usually adopted in deductive database systems, as Datalog, for practical reasons. Other works add different syntactic conditions as [4], where the notion of *guarded negation* is adapted to database queries both for SQL and Datalog languages in order to improve performance. In the case of Datalog, guarded negation introduces an additional syntactical condition to stratification. Although it is computationally well-behaved and subsumes several well-known query languages as unions of conjunctive queries, monadic Datalog and frontier-guarded tuple-generating dependencies, it is actually subsumed by stratified Datalog. As $HH_{\neg}(\mathcal{C})$, in turn, subsumes stratified Datalog, it also subsumes Datalog with guarded negation.

In addition, the syntactic limitations associated to the stratification approach can be overcome in practical situations of potentially non-stratifiable programs, which can be modeled by equivalent stratifiable databases. We illustrate this point by an example showed in [21] and also related in [52] as a classical example of non-stratifiable program, that can be tackled both with the Well-Founded semantics and Stable Models.

Example 1. In this example it is shown a general scheme for a two-people game with a finite space of states. This scheme allows to determine the winning states of the game (those that guarantee the victory for the player in turn) by means of one single clause reflecting a simple idea: one wins if the opponent cannot win because it cannot move. The clause is:

$$\forall x \forall y\ winning(x) \Leftarrow move(x, y) \wedge \neg winning(y),$$

where *move* is defined for the concrete game. This program is not stratifiable due the negative cycle in *winning*. Nevertheless, it is easy to see that the winning strategy is based on the parity of the number of movements. Using that fact, it is straightforward to encode this strategy as follows:

$$\forall x \forall y\ canMove(x) \Leftarrow move(x, y),$$

$$\forall x \forall y\ possibleWinning(x) \Leftarrow oddMove(x, y) \wedge \neg canMove(y),$$

$$\forall x \forall y\ winning(x) \Leftarrow move(x, y) \wedge \neg possibleWinning(y).$$

Here $oddMove(x, y)$ represents a change from state x to state y in an odd number of movements and it is defined as:

$$\forall x \forall y\ oddMove(x, y) \Leftarrow move(x, y),$$

$$\forall x \forall y \forall z_1 \forall z_2\ oddMove(x, y) \Leftarrow move(x, z_1) \wedge move(z_1, z_2) \wedge oddMove(z_2, y).$$

This program represents a stratifiable database which is suitable for $HH_\neg(\mathcal{C})$, and even for Datalog. The definition of the predicate *oddMove* can be simplified in $HH_\neg(\mathcal{C})$, making use of constraints. In addition, our language combines implication, negation and recursion, so more complex queries can be formulated as, for instance:

$$move(a, b) \Rightarrow winning(x),$$

that asks for the winner positions, assuming the existence of the new movement $move(a, b)$. □

One of the main advantages of our language is the use of hypothetical queries, an unusual feature in a database language. However, there exist some works as Hypothetical Datalog [9,10], that is an extension of Horn-clause logic allowing hypothetical queries in a similar way to our proposal. Queries are allowed to include the local (hypothetical) addition and deletion of tuples from the database. In both, additions (`A ← B [add : C]`) and deletions (`A ← B [del : C]`), the atomic term `C` is temporarily added or deleted from the extensional database in order to solve the query `B`, which can be understood as a dynamic modification of the database, similarly to the $HH_\neg(\mathcal{C})$ behavior. Nevertheless, this is a very restrictive form of hypothetical queries, which implies a great simplification of the approach, as $HH_\neg(\mathcal{C})$ allows complete clauses as the antecedent in hypothetical queries, i.e., it allows to dynamically change the intensional database (which corresponds to the formalization within the intuitionistic logic). For instance, in the previous example, the query $move(a, b) \Rightarrow winning(x)$ can be formulated also in Hypothetical Datalog. But, it is not possible to assume a more general rule for the predicate *move*, that defines the possible movements of the game. In contrast, for instance, the formula $\forall x \forall y (2 * x \approx y \Rightarrow move(x, y)) \Rightarrow winning(z)$ is a valid query in $HH_\neg(\mathcal{C})$. Moreover, when comparing Hypothetical Datalog and its theoretical foundations to $HH_\neg(\mathcal{C})$, we must emphasize that our logic combines connectives and constraints. In [15] another approach for hypothetical queries, also including positive and negative hypotheses, is proposed, but neither negation nor implications are allowed in clauses.

Previously, we pointed out the difficulty of dealing with negation in logic programming. The semantics of negation is even more complex in the presence of implication in goals, as it is the case of Hypothetical Datalog. Additional obstacles arise when considering the logic *HH*, due to the inclusion of universal quantifiers. There exist several proposals aimed to introduce negation in *HH* (see for instance [24,37,17]). [17] is the first work introducing negation as failure into N-Prolog. In [37] a natural calculus is proposed for *HH* with negation. [24] is closer to our approach in the sense that a sequent calculus as well as a fixpoint semantics is defined for this logic. Instead of stratification, in order to preserve monotonicity, two forcing relations (positive and negative) are introduced, and in addition completely and incompletely defined predicates must be distinguished. In practice, these issues yield to a hard computation of the fixpoint in a concrete implementation.

With respect to constraint database systems, during the last couple of decades, constraint databases have received much attention because they are specially suited for applications subject to geometric interpretation, notably in geographic information systems (GIS), spatial databases and spatio-temporal databases [22,44,42,30]. Current trends are oriented to develop efficient operators and indexing of geometric data. Specific academic systems include MLPQ/GIS [28], DISCO [11] and PReSTO [43,12]. Output from this research was transferred to commercial systems and nowadays, several database vendors offer constraint-based databases as IBM DB2, MS SQL Server, Oracle, PostgreSQL, mainly for GIS (Geographic Information System) applications. However, as in relational algebra, negation only occurs in difference operators, so that negated predicates cannot be queried, as we do allow. Negation in the specific field of *CDDB* systems has been also studied [23,42]. In [23], different uses of negation are identified and the more general one requires delaying for constraints to become ground, which is a severe restriction. The treatment of negation of [42] corresponds to stratified Datalog for safe constraint queries. In our language, negation is even more complex due to the presence of implication and universal quantification in goals.

1.2. Contributions and relationship to prior work

In this paper we give a complete picture of the $HH_\neg(\mathcal{C})$ language as an expressive constraint deductive database language, including its theoretical foundation, as well as the description of a prototype system based on it. The paper provides an integrating view of previous works related to this language, [38,2], and extends them in several aspects:

- In [38] the programming scheme $HH_\neg(\mathcal{C})$ was formalized defining a proof-semantics and a stratified fixpoint semantics, that is sound and complete w.r.t. the former. Here, we show in detail these two formalizations which support the scheme, emphasizing the purpose of the fixpoint semantics as an operational semantics that guides the implementation. In addition, we include full proofs for the equivalence results.
- In [2] we presented a first Prolog prototype implementing that scheme, based on the fixpoint semantics, that is independent on the particular constraint system. This prototype has been enhanced with different improvements. In this paper we show a detailed description of the improved $HH_\neg(\mathcal{C})$ system, the way in which technical problems have been

solved, and some selected examples evincing the usefulness of the scheme. The most relevant features added to the system in the present work are:

- We have investigated how to incorporate aggregate functions to the language. The most usual aggregate operations are integrated in the language as part of the constraint system, in such a way that the computation of aggregate functions is delegated to the constraint solver. This is a non-trivial issue in our context that has been solved by taking advantage of our stratification and dependency graph notions. With this aim, the dependency graph specification has been enriched by adding dependencies due to aggregate operations.
- We have also improved the constraint systems, which are now able to deal with a limited form of constraint combination. Specifically, the system can deal with more complex constraints because the generic interface of the constraint solvers can identify constraint belonging to different domains, split them, and send each part of the initial constraint to its corresponding domain solver.
- We have improved the computation of queries, avoiding the recomputation of the whole database from scratch, in cases where stratification changes.
- Finally, we have design a better and more complete user interface, also enhancing its performance.

1.3. Organization of the paper

The rest of the paper is organized as follows. In Section 2, $HH_{\neg}(\mathcal{C})$ is informally presented as a query language by means of examples. In Section 3, the syntax of the language is formally introduced and concrete examples, which illustrate its potential and expressiveness, are shown. In Section 4, the proof-theoretic semantics for $HH_{\neg}(\mathcal{C})$ is defined as a foundation for the scheme. In Section 5, a fixpoint semantics, based on the notion of stratification, is presented and proved to be sound and complete with respect to the proof-theoretic semantics. Section 6 introduces a user-oriented description of the current system and the aggregate functions, and schematizes the computation stages of the system. Section 7 is devoted to the description of constraint domains and the implementation of the corresponding solvers. Aggregate functions are introduced as a part of the constraint language. Section 8 is an overview of the fixpoint semantics implementation, making emphasis on the implementation of the forcing relation which supports the semantics of $HH_{\neg}(\mathcal{C})$. In particular, the difficulties that have been overcome to implement the forcing of the implication are explained. Section 9 summarizes some conclusions and sketches future works. In Appendix A we include the full proofs for the results belonging to Section 4. Appendix B describes a form of the dependency graph needed to implement the forcing of the implication and constraints including aggregates. Finally, in Appendix C, the implementation of the constraint systems is provided, and Appendix D includes implementation details for the forcing relation.

2. $HH(\mathcal{C})$ as a database language: A first glance

This section is devoted to informally present $HH(\mathcal{C})$ as a suitable language for constraint databases. In our system, a database is a logic program: a set of clauses. Facts (ground atoms) define the extensional database, and clauses with body define the intensional database. The last ones can be seen as the definition of views in relational databases. The evaluation of a query with respect to a deductive database can be seen as the computation of a goal (query) from a program (database), and the answer is a constraint.

2.1. Relations and predicates

The relational model deals with (finite) relations which can be defined both extensionally, as sets of tuples, and intensionally, by means of views. A relation has a name, an arity, and its meaning may be understood as a set of tuples. As well, a predicate has a name, an arity, and its meaning can be understood as a set of constraints over its arguments. Predicates can also be defined both extensionally, by means of facts, and intensionally, by means of clauses.

Example 2. Fig. 1 defines some relations extensionally (*client* and *mortgageQuote*), and intensionally (*accounting*) both in *RA* and in $HH(\mathcal{C})$. Extensional relations are defined as tables in the relational model in (a) and as extensional predicates in $HH(\mathcal{C})$ (facts of (c)). The relation *accounting* is defined as a view in the relational model in (b), and as an intensional predicate in $HH(\mathcal{C})$ (clauses with a non-empty right-hand side of (c)).

In *RA* the result of computing the view *accounting* is the relation:

name	*salary*	*quote*
brown	1500	400
mcandrew	3000	100

accounting

In $HH(\mathcal{C})$ this query corresponds to the computation of the goal $accounting(n, s, q)$ whose answer is $(n \approx brown \wedge s \approx 1500 \wedge q \approx 400) \vee (n \approx mcandrew \wedge s \approx 3000 \wedge q \approx 100)$. This introductory example shows some relations that could also be

(a) Relations extensionally defined as relational tables:

name	*balance*	*salary*
smith	2000	1200
brown	1000	1500
mcandrew	5300	3000

client

name	*quote*
brown	400
mcandrew	100

mortgageQuote

(b) A relation defined as a relational view:

$$accounting \leftarrow \pi_{name,salary,quote}(\sigma_{quote \geqslant 100}(client \bowtie mortgageQuote))$$

(c) The above relations defined as an $HH(\mathcal{C})$ program:

$client(smith, 2000, 1200).$ $\quad mortgageQuote(brown, 400).$
$client(brown, 1000, 1500).$ $\quad mortgageQuote(mcandrew, 100).$
$client(mcandrew, 5300, 3000).$

$$\forall name \forall salary \forall quote \forall balance(accounting(name, salary, quote) \Leftarrow client(name, balance, salary) \wedge mortgageQuote(name, quote) \wedge quote \geqslant 100).$$

Fig. 1. Relations vs. $HH(\mathcal{C})$ predicates.

computed in Datalog with constraints [27], but it will be extended later in Example 6 with some new relations that exceed the capabilities of such a system. □

2.2. Infinite data as finite representations

One of the advantages of using constraints in the (general) context of *LP* is that they provide a natural way for dealing with infinite data collections using finite (intensional) representations. Constraint databases [30] have inherited this feature. We illustrate this point with the following example.

Example 3. Assume the instance $HH(\mathcal{R})$, i.e., the domain of arithmetic constraints over real numbers. We are interested in describing regions in the plane. A region is a set of points identified by its characteristic function (a Boolean function which evaluates to *true* over the points of such a region, and to *false* over the rest of points of the plane). For example, a rectangle can be determined by its left-bottom corner (x_1, y_1) and its right-top corner (x_2, y_2) and its characteristic function can be expressed by the next clause:

$$\bar{\forall} rectangle(x_1, y_1, x_2, y_2, x, y) \Leftarrow x \geqslant x_1 \wedge x \leqslant x_2 \wedge y \geqslant y_1 \wedge y \leqslant y_2,$$

where $\bar{\forall}$ represents the universal closure of a formula. Analogously, $\bar{\exists}$ will represent the existential closure.

Notice that a rectangle contains (in general) an infinite set of points and they are finitely represented in an easy way by means of real constraints. From a database perspective, this is a very interesting feature: databases were conceived to work with finite pieces of information, but introducing constraints makes it possible to manage (potentially) infinite sets of data.

The goal $rectangle(0, 0, 4, 4, x, y) \wedge rectangle(1, 1, 5, 5, x, y)$ computes the intersection of two rectangles and an answer can be represented by the constraint:

$$(x \geqslant 1) \wedge (x \leqslant 4) \wedge (y \geqslant 1) \wedge (y \leqslant 4).$$

A circle can be defined by its center and radius, using non-linear constraints now:

$$\bar{\forall} circle(xc, yc, r, x, y) \Leftarrow (x - xc)^2 + (y - yc)^2 \leqslant r^2.$$

We can ask, for instance, whether any pair (x, y) such that $x^2 + y^2 = 1$ (the circumference centered in the origin and radius 1) is inside the circle with center $(0, 0)$ and radius 2 by means of the goal:

$$\bar{\forall}\big(x^2 + y^2 \approx 1 \Rightarrow circle(0, 0, 2, x, y)\big).$$

This goal is not expressible in standard deductive database languages because, in addition to constraints, it involves universal quantifiers and implication. Even Hypothetical Datalog cannot deal with this goal due to the universal quantifiers. These components constitute a big step in expressivity. □

Since $HH(\mathcal{C})$ does not support negation, it is not still complete w.r.t. *RA* as we show next.

- Projection. $E = \pi_{i_1}, \ldots, \pi_{i_k}(E_1)$

  $\bar{\forall} e(x_{i_1}, \ldots, x_{i_k}) \Leftarrow e_1(x_1, \ldots, x_n)$.

- Selection. $E = \sigma_{t_1 \theta t_2}(E_1)$

  $\bar{\forall} e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \wedge C_\theta$.

- Cartesian product. $E = E_1 \times E_2$

  $\bar{\forall} e(x_1, \ldots, x_n, x_{n+1}, \ldots, x_m) \Leftarrow e_1(x_1, \ldots, x_n) \wedge e_2(x_{n+1}, \ldots, x_m)$.

- Set union. $E = E_1 \cup E_2$

  $\bar{\forall} e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \vee e_2(x_1, \ldots, x_n)$.

- Set difference. $E = E_1 - E_2$

  $\bar{\forall} e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \wedge \neg e_2(x_1, \ldots, x_n)$.

E and E_i (resp.) are relational expressions represented as e and e_i (resp.) predicates. C_θ is the constraint corresponding to the condition $t_1 \theta t_2$.

Fig. 2. Relational operators as $HH\neg(\mathcal{C})$ programs.

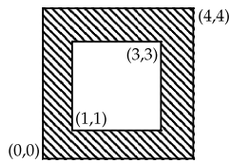


Fig. 3. Regions in the plane.

2.3. Need for negation

What a database user might want is to have the basic relational operations available in this language. In fact, a database language is complete w.r.t. *RA* if these operations can be expressed within the language. As it is shown in Fig. 2, $HH(\mathcal{C})$ can express projection, Cartesian product, union, and selection. For the last one, it is required that the constraint system $\mathcal{C}$ incorporates (or can express) the operators θ, in order to build the corresponding constraint C_θ. For instance, $\sigma_{\$i \leqslant \$j}$ corresponds to $x_i \leqslant x_j$. As we will see in Section 3.1, any constraint system in our scheme satisfies this requirement. But expressing set difference needs some kind of negation. So we have added the connective $\neg$ to $HH(\mathcal{C})$, obtaining a complete database language which will be formalized in the next section. There are some other situations, besides relational database requirements, in which negation is needed.

Example 4. Returning to Example 3, we define the dashed frame depicted in Fig. 3 by the inner region of a large rectangle and the outer region of a small rectangle with the goal

$$rectangle(0, 0, 4, 4, x, y) \wedge \neg rectangle(1, 1, 3, 3, x, y),$$

and an answer can be represented by the constraint:

$$(y > 3 \wedge y \leqslant 4 \wedge x \geqslant 0 \wedge x \leqslant 4) \vee (y \geqslant 0 \wedge y < 1 \wedge x \geqslant 0 \wedge x \leqslant 4) \vee$$
$$(y \geqslant 0 \wedge y \leqslant 4 \wedge x > 3 \wedge x \leqslant 4) \vee (y \geqslant 0 \wedge y \leqslant 4 \wedge x \geqslant 0 \wedge x < 1).$$

In this example, we assume that negation can be effectively handled by the constraint solver, an issue addressed later in this paper. □

3. The language $HH_\neg(\mathcal{C})$

The formalisms which $HH(\mathcal{C})$ is founded on [31,19] do not support any kind of negation, so the language is not expressive enough in the field of database systems. We have extended $HH(\mathcal{C})$ including negation to obtain a *CDDB* language which is complete w.r.t. *RA*. In this section, we make precise the syntax of the formulas of $HH(\mathcal{C})$ extended with negation, denoted by $HH_\neg(\mathcal{C})$; next we introduce more database examples.

3.1. Syntax

As usual, to build the syntactic objects of the logic, we consider a set of variables and a signature containing:

- defined predicate symbols, representing the names of database relations, to build atoms,
- non-defined (built-in) predicate symbols, including at least the comparison $\leqslant$ and the equality predicate symbol $\approx$, to build atomic constraints, and
- constant and operator symbols, which depend on the particular constraint system, to build terms.

Since our interest is to represent databases, we only use finite signatures.

Well-formed formulas in $HH\neg(\mathcal{C})$ can be classified into clauses D (defining database relations) and goals (or queries) G. They are recursively defined by the following rules:

$$D ::= A \mid G \Rightarrow A \mid D_1 \wedge D_2 \mid \forall x D,$$

$$G ::= A \mid \neg A \mid C \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \Rightarrow G \mid C \Rightarrow G \mid \exists x G \mid \forall x G.$$

A represents an atom, i.e., a formula of the form $p(t_1, \ldots, t_n)$, where p is a defined predicate symbol of arity n, and t_i are terms; C represents a constraint. The incorporation of negated atoms in goals is the addition to $HH(\mathcal{C})$. Negation is not allowed in the head of a clause, but inside its body.

3.1.1. The constraint system $\mathcal{C}$

The constraints we consider belong to a generic system $\mathcal{C} = \langle \mathcal{L}_\mathcal{C}, \vdash_\mathcal{C} \rangle$, where $\mathcal{L}_\mathcal{C}$ is the constraint language and $\vdash_\mathcal{C}$ is a binary *entailment relation*. $\Gamma \vdash_\mathcal{C} C$ denotes that the constraint C is inferred in the constraint system $\mathcal{C}$ from the set of constraints Γ. Some minimal conditions are imposed to $\mathcal{C}$ to be a constraint system:

- $\mathcal{L}_\mathcal{C}$ contains at least every first-order formula built up using:
  - $\top$ (*true*), $\bot$ (*false*),
  - built-in predicate symbols,
  - the connectives $\wedge, \neg$, and the existential quantifier $\exists$.
- Regarding to $\vdash_\mathcal{C}$:
  - It includes inference logic rules for the considered connectives and quantifiers.
  - It is *compact*, i.e., $\Gamma \vdash_\mathcal{C} C$ implies that there exists a finite set $\Gamma' \subseteq \Gamma$, such that $\Gamma' \vdash_\mathcal{C} C$.
  - It is *closed under substitution*, i.e., $\Gamma \vdash_\mathcal{C} C$ implies $\Gamma\sigma \vdash_\mathcal{C} C\sigma$ for every substitution σ.

Let us remark that $\mathcal{C}$ is required to deal with negation, because the incorporation of the connective $\neg$ to the language HH yields to the need for incorporating the negation in the constraint system, which has the responsibility of checking the satisfiability of answers in the constraint domain.

We say that a constraint C is $\mathcal{C}$-satisfiable if $\emptyset \vdash_\mathcal{C} \bar{\exists} C$, where $\bar{\exists} C$ stands for the existential closure of C. C and C' are $\mathcal{C}$-equivalent if $C \vdash_\mathcal{C} C'$ and $C' \vdash_\mathcal{C} C$.

The constraint systems of the previous examples verify the required minimal conditions aforementioned. Moreover, they also include the connective $\vee$ (as usual), constants to represent numbers and names, arithmetical operators, and more built-in predicates $(>, \geqslant, \ldots)$.

For instance, for the constraint system $\mathcal{R}$ of Real-closed Fields, $\mathcal{L}_\mathcal{R}$ is a first-order language with all classical logical connectives including negation, and $\Gamma \vdash_\mathcal{R} C$ holds when $Ax_\mathcal{R} \cup \Gamma \vdash_\approx C$, where $Ax_\mathcal{R}$ is Tarski's axiomatization of the real numbers, and $\vdash_\approx$ is the entailment relation of classical logic with equality. An example of a concrete constraint is $\neg(x \approx 0.2)$, also written as $x \not\approx 0.2$ for the sake of simplicity.

It is easy to see that every formula allowed in the selection operation of *RA* can be expressed with an equivalent constraint, since for any $\mathcal{C}$, the language $\mathcal{L}_\mathcal{C}$ contains the built-in predicates $\leqslant$ and $\approx$, and the connectives $\wedge$ and $\neg$.

3.1.2. $HH\neg(\mathcal{C})$ programs

Programs, denoted by Δ, are sets of clauses and represent databases. As usual in Logic Programming, they can still be viewed as sets of implicative formulas with atomic head, in the way we precise now. The *elaboration* of a program Δ is the set $elab(\Delta) = \bigcup_{D \in \Delta} elab(D)$, where $elab(D)$ is defined by:

- $elab(A) = \{\top \Rightarrow A\}$,
- $elab(G \Rightarrow A) = \{G \Rightarrow A\}$,
- $elab(\forall x D) = \{\forall x D' \mid D' \in elab(D)\}$,
- $elab(D_1 \wedge D_2) = elab(D_1) \cup elab(D_2)$.

So, elaborated clauses are formulas of the form $\forall x_1 \ldots \forall x_n (G \Rightarrow A)$ (or simply $\forall \bar{x}(G \Rightarrow A)$), but notice that clauses inside G are not required to be elaborated. The use of the elaborated form, instead of general $HH\neg(\mathcal{C})$ clauses, has some practical benefits:

- It permits to specify a database view that defines a predicate (database relation) p by means of a set of elaborated clauses whose heads are atoms beginning with the predicate symbol p, as it is done in logic programs with Horn clauses.

- It permits to define a calculus governing $HH\neg(\mathcal{C})$ without rules introducing connectives in the left, providing the uniformity property of the calculus, which guarantees completeness of goal-oriented search for proofs. Notice that in the calculus $\mathcal{UC}_\neg$ (introduced in Section 4) there is only the rule (*Clause*) to deal with atomic goals, that corresponds to the SLD-Resolution rule of logic programming.
- It facilitates the formalization of the fixpoint operator used to define the fixpoint semantics introduced in Section 5.2.

For convenience, we will also use the common notation $\forall x_1 \ldots \forall x_n (A \Leftarrow G)$ as in previous examples.

3.2. Examples of $HH_\neg(\mathcal{C})$

Once we have formalized the syntax of our language, we introduce more examples showing the advantages of our proposal w.r.t. other common database languages. As an important benefit of our approach, we stress the ability to formulate hypothetical and universally quantified queries. In addition, variables can be explicitly and existentially quantified in queries avoiding the computation of an explicit answer for these variables.

In these examples, the instance $HH_\neg(\mathcal{FR})$ is used, where $\mathcal{FR}$ is a hybrid constraint system which combines constraints over finite and real numbers domains, ensuring domain independence. Instantiating the scheme with mixed constraint systems will be very useful in the context of databases. In [18], a hybrid constraint system subsuming $\mathcal{FR}$ is presented.

Example 5. Consider the following travel database. The predicate *flight*(*Origin*, *Destination*, *Time*) represents an extensional database relation of direct flights from *Origin* to *Destination* and duration *Time*:

$$flight(mad, par, 1.5),$$

$$flight(par, ny, 10),$$

$$flight(london, ny, 9).$$

In turn, *travel*(*Origin*, *Destination*, *Time*) represents an intensional database relation, expressing that it is possible to travel from *Origin* to *Destination* in a time greater or equal than *Time*, possibly concatenating some flights:

$$\bar{\forall} travel(x, y, t) \Leftarrow flight(x, y, w) \wedge t \geqslant w,$$

$$\bar{\forall} travel(x, y, t) \Leftarrow flight(x, z, t_1) \wedge travel(z, y, t_2) \wedge t \geqslant t_1 + t_2.$$

The next goal asks for the duration of a flight from Madrid to London in order to be able to travel from Madrid to New York in 11 hours at most:

$$flight(mad, london, t) \Rightarrow travel(mad, ny, 11).$$

The answer constraint of this query will be $11 \geqslant t + 9$ which is $\mathcal{FR}$-equivalent to the final answer $t \leqslant 2$.

Another hypothetical query to the previous database is the question that if it is possible to travel from Madrid to some place in any time greater than 1.5. The goal $\forall t(t > 1.5 \Rightarrow \exists y\ travel(mad, y, t))$ includes also universal quantification, and the corresponding answer is $\top$.

We can also compare $HH_\neg(\mathcal{C})$ to relational calculus (whose underlying logic is richer than the used on implemented *CDDB* languages). For instance, the query $\neg(\exists t\ flight(x, y, t)) \wedge x \not\approx y$, or its equivalent $(\forall t \neg flight(x, y, t)) \wedge x \not\approx y$, which represents the cities in the database that have no direct flights between them, is not safe in the domain relational calculus, because it contains a negated formula whose free variables are not limited. This problem is avoided in our system because formulas are interpreted in the context of the constraint domain of the particular instance and no test for this kind of safety is needed. In fact, $(\forall t \neg flight(x, y, t)) \wedge x \not\approx y$ represents a valid $HH_\neg(\mathcal{FR})$ query, which has as answer constraint:

$$(x \not\approx mad \vee y \not\approx par) \wedge (x \not\approx par \vee y \not\approx ny) \wedge (x \not\approx lon \vee y \not\approx ny)$$

in the domain of the cities registered in the current database. However, the query is not allowed in Datalog with constraints due, in this case, to the quantifier occurrence.

Assume now a more realistic situation in which flight delays may happen, which is represented by the following definition:

$$\bar{\forall} deltravel(x, y, t) \Leftarrow flight(x, y, t_1) \wedge delay(x, y, t_2) \wedge t \geqslant t_1 + t_2,$$

$$\bar{\forall} deltravel(x, y, t) \Leftarrow flight(x, z, t_1) \wedge delay(x, z, t_2) \wedge deltravel(z, t, t_3) \wedge t \geqslant t_1 + t_2 + t_3.$$

Tuples of *delay* may be in the extensional database or may be assumed when the query is formulated. For instance, the query

$$\forall x\big(delay(par, x, 1) \wedge delay(mad, par, 0.5)\big) \Rightarrow deltravel(mad, ny, t)$$

represents the query: What is the time needed to travel from Madrid to New York assuming that for any destination there is a delay of one hour from Paris, and the flight from Madrid to Paris is half an hour delayed? According to its proof-theoretic interpretation, the clause $\forall x(delay(par, x, 1) \wedge delay(mad, par, 0.5))$ will be added locally to the database to solve the goal $deltravel(mad, ny, t)$, and it will be discarded after the computation as they are hypothetical assumptions. Since flights may or may not be delayed, a more general view can be defined in order to know the expected time of a trip:

$$\bar{\forall} trip(x, y, t) \Leftarrow nondeltravel(x, y, t) \vee deltravel(x, y.t),$$

$$\bar{\forall} nondeltravel(x, y, t) \Leftarrow travel(x, y, t) \wedge \neg delayed(x, y),$$

$$\forall x \forall y\ delayed(x, y) \Leftarrow \exists t\ deltravel(x, y, t).$$

Notice that the last formula is equivalent to $\bar{\forall} delayed(x, y) \Leftarrow deltravel(x, y, t)$. Since explicit existential quantifiers are allowed in $HH\neg(\mathcal{C})$, they can also be used to improve readability and facilitate the specification of some predicates. □

Example 6. In this example we extend the database for a bank introduced in Example 2. The extensional database is given by facts for the relations $client(Name, Balance, Salary)$, $mortgageQuote(Name, Quote)$, $pastDue(Name, Amount)$ and $branch(Office, Name)$ as follows:

$client(smith, 2000, 1200).$ $mortgageQuote(brown, 400).$

$client(brown, 1000, 1500).$ $mortgageQuote(mcandrew, 100).$

$client(mcandrew, 5300, 3000).$

$pastDue(smith, 3000).$ $branch(lon, smith).$

$pastDue(mcandrew, 100).$ $branch(mad, brown).$

$branch(par, mcandrew).$

As an additional restriction we assume that each client has, at most, one mortgage quote. Next, we introduce some views defining the intensional part of the database. The first one captures the clients that have a mortgage: a client has a mortgage if there exists a mortgage quote associated to him:

$$\bar{\forall} hasMortgage(x) \Leftarrow mortgageQuote(x, y).$$

A debtor is a client who has a past due with an amount greater than his balance:

$$\bar{\forall} debtor(x) \Leftarrow client(x, y, z) \wedge pastDue(x, w) \wedge w > y.$$

The applicable interest rate to a client is specified by the next relation:

$$\bar{\forall} interestRate(x, 2) \Leftarrow client(x, y, z) \wedge y < 1200,$$

$$\bar{\forall} interestRate(x, 5) \Leftarrow client(x, y, z) \wedge y \geqslant 1200.$$

The next relation $newMortgage(Name, Quote)$ specifies that a non-debtor client $Name$ can be given a new mortgage with $Quote$ in two situations. First, if he has no mortgage, a mortgage quote smaller than 40% of his salary can be given. And, second, if he has a mortgage quote already, then the sum of this quote and the new one has to be smaller than that percentage:

$$\bar{\forall} newMortgage(x, w) \Leftarrow client(x, y, z) \wedge \neg debtor(x) \wedge \neg hasMortgage(x) \wedge w \leqslant 0.4 * z,$$

$$\bar{\forall} newMortgage(x, w) \Leftarrow client(x, y, z) \wedge \neg debtor(x) \wedge mortgageQuote\big(x, w'\big) \wedge w + w' \leqslant 0.4 * z,$$

$$\bar{\forall} gotMortgage(x) \Leftarrow newMortgage(x, w).$$

If the client satisfies the requirements to be given a new mortgage, then it is possible to apply for a personal credit, whose amount is smaller than 6000. Otherwise, if such a client does not satisfy that requirements, the amount must be between 6000 and 20,000. The relation $personalCredit(Name, Amount)$ formalizes these conditions:

$$\bar{\forall} personalCredit(x, y) \Leftarrow \big(gotMortgage(x) \wedge y < 6000\big) \vee \big(\neg gotMortgage(x) \wedge y \geqslant 6000 \wedge y < 20{,}000\big).$$

Moreover, it is possible to define a view with the quote and the salary of clients whose mortgage quote is greater than 100 with the following relation *accounting*(*Name*, *Salary*, *Quote*) which corresponds to the predicate *accounting* of Example 2:

$$\bar{\forall} accounting(x, z, w) \Leftarrow client(x, y, z) \wedge mortgageQuote(x, w) \wedge w \geqslant 100.$$

The previous predicates define the database that we are going to use for illustrating some queries. As a first example, we can query whether every client is a debtor:

$$\forall x debtor(x),$$

for which the answer is $\bot$.

For knowing whether there are debtors with a past due amount greater than 1000, the following query can be formulated:

$$\exists x \exists y\ debtor(x) \wedge pastDue(x, y) \wedge y > 1000,$$

and the answer is $\top$. Note that we are using quantifiers for variables x and y, meaning that there are no explicit conditions over them. Otherwise, the answer will be a constraint over such variables.

The next query corresponds to the question: If for a non-specific client we assume that has a balance greater than 2000, what would the interest rate be?

$$\forall x \exists y \exists z \big(client(x, y, z) \Rightarrow \big(y > 2000 \Rightarrow interestRate(x, w)\big)\big).$$

Here we are using nested implication to formulate a hypothetical query. The answer is the constraint $w \approx 5$.

The next query involves negation and can be read as: which clients can get a mortgage quote of 400 but *not* a personal credit?

$$newMortgage(x, 400) \wedge \neg personalCredit(x, y),$$

and the answer is the constraint $x \approx mcandrew \wedge y \geqslant 6000 \wedge y < 20{,}000$, which means that it is possible to give a new mortgage to the client McAndrew, but it is *not* allowed to give him a personal credit of an amount between 6000 and 20,000. □

4. Proof-theoretic semantics

Several kinds of semantics have been defined for $HH(\mathcal{C})$ without negation, including proof-theoretic, operational [31] and fixpoint semantics [19], as well as for its higher-order version [33]. The proof-theoretic and fixpoint approaches have been adapted in order to formalize the extension $HH_{\neg}(\mathcal{C})$. In addition we have proven that they are equivalent.

The simplest way for explaining the meaning of programs and goals in the present framework is by using a proof-theoretic semantics. Queries formulated to a database are interpreted by means of the inference system which governs the underlying logic. This proof-system, called $\mathcal{UC}_{\neg}$ (Uniform calculus handling Constraints and negation) is a sequent calculus that combines traditional inference rules with the entailment relation $\vdash_{\mathcal{C}}$ of the generic constraint system $\mathcal{C}$.

Sequents have the form $\Delta; \Gamma \vdash G$, where programs Δ and sets of constraints Γ are on the left, and goals on the right. The notation $\Delta; \Gamma \vdash_{\mathcal{UC}_{\neg}} G$ means that the sequent $\Delta; \Gamma \vdash G$ has a proof using the rules of $\mathcal{UC}_{\neg}$. A proof of a sequent is a finite tree whose root is the sequent to be proved and the nodes are sequents. The rules regulate relationship between child nodes and parent nodes and the leaves are nodes of the form $\Gamma \vdash_{\mathcal{C}} C$. If $\Delta; C \vdash_{\mathcal{UC}_{\neg}} G$, then C is called an *answer constraint* to the query G in the database Δ, that can be understood as the meaning of the query G formulated to the database Δ. The idea is that G is true for Δ if the constraint C is satisfied. $\mathcal{UC}_{\neg}$ carries out only uniform proofs in the sense defined by Miller et. al. [35], i.e., goal-oriented proofs. The rules are applied backwards and, at any step, only one rule of the calculus can be applied, that is the corresponding to the structure of the goal. Notice that we are assuming that any constraint will be treated as a whole, and the only applicable rule in this case is (C). (*Clause*) is used for atoms beginning with defined predicate symbols, the rest of the rules correspond to the outermost connective/quantifier of the goal (non-constraint) to be proved.

This proof system is an extension of the calculus $\mathcal{UC}$, introduced in [31] which provided proof-theoretic semantics for $HH(\mathcal{C})$. The incorporation of negation to the language makes it necessary to extend the notion of derivability, because there is no rule for this connective in $\mathcal{UC}$. Therefore, $\mathcal{UC}_{\neg}$ incorporates a new rule ($\neg$) to formalize derivability of negated atoms. The rules defining the extended calculus appear in Fig. 4.

Next, we explain the rules ($\exists$), (*Clause*) and ($\neg$), the others correspond to widespread intuitionistic rules introducing connectives on the right of the sequent (see, e.g., [35]), except (C) which deals with goals that are pure constraints.

($\exists$) captures the fact that the witness in the proof of an existentially quantified formula can be represented by a constraint which can be more general than an equality $x \approx t$ simulating a substitution (e.g., $(x * x \approx 2)$ represents the witness $\sqrt{2}$, which cannot be written as a term).

(*Clause*) represents backchaining and allows one to prove an atomic goal $A \equiv p(t_1, \ldots, t_n)$, where p is a defined predicate symbol, using a program clause whose head $A' \equiv p(t'_1, \ldots, t'_n)$ is not required to unify with A, but rather solving a new

$$\frac{\Gamma \vdash_{\mathcal{C}} C}{\Delta;\Gamma \vdash C}\ (C) \qquad \frac{\Delta;\Gamma \vdash \exists x_1 \ldots \exists x_n((A' \approx A) \wedge G)}{\Delta;\Gamma \vdash A}\ (\mathit{Clause})(*), \text{ where } \forall x_1 \ldots \forall x_n(G \Rightarrow A') \text{ is a variant of a formula of } \mathit{elab}(\Delta)$$

$$\frac{\Delta;\Gamma \vdash G_i}{\Delta;\Gamma \vdash G_1 \vee G_2}\ (\vee)\ (i=1,2) \qquad \frac{\Delta;\Gamma \vdash G_1 \ \Delta;\Gamma \vdash G_2}{\Delta;\Gamma \vdash G_1 \wedge G_2}\ (\wedge)$$

$$\frac{\Delta, D;\Gamma \vdash G}{\Delta;\Gamma \vdash D \Rightarrow G}\ (\Rightarrow) \qquad \frac{\Delta;\Gamma, C \vdash G}{\Delta;\Gamma \vdash C \Rightarrow G}\ (\Rightarrow_C)$$

$$\frac{\Delta;\Gamma, C \vdash G[y/x] \ \Gamma \vdash_{\mathcal{C}} \exists y C}{\Delta;\Gamma \vdash \exists x G}\ (\exists)(**) \qquad \frac{\Delta;\Gamma \vdash G[y/x]}{\Delta;\Gamma \vdash \forall x G}\ (\forall)(**)$$

$$\frac{\Gamma \vdash_{\mathcal{C}} \neg C \text{ for every } \Delta; C \vdash A}{\Delta;\Gamma \vdash \neg A}\ (\neg)$$

(*) $x_1, \ldots, x_n$ fresh for A
(**) y fresh for the formulas in the conclusion of the rule

Fig. 4. Rules of the sequent calculus $\mathcal{UC}_{\neg}$.

existentially quantified goal that, by applying the $(\exists)$ rule, will result in a search for a constraint that implies the equality $A' \approx A$ (this stands for $t'_1 \approx t_1 \wedge \cdots \wedge t'_n \approx t_n$).

The idea of interpreting the query $\neg A$ from a database Δ, by means of an answer constraint C is that, whenever C' is a possible answer to the query A from Δ, then $C \vdash_{\mathcal{C}} \neg C'$. This is formalized with $(\neg)$. We say that $(\neg)$ is a *metarule* since its premise considers any derivation $\Delta; C \vdash A$ of the atom A. In practice, there is a derivation of $\neg A$ when the set of answer constraints of A from Δ is finite.

Next we show two examples of proof derivation trees.

Example 7. Consider a fragment of the travel database of Example 5.

$$\text{Let } \Delta = \big\{ \mathit{flight}(\mathit{mad}, \mathit{par}, 1.5),\ \mathit{flight}(\mathit{par}, \mathit{ny}, 10),\ \mathit{flight}(\mathit{lon}, \mathit{ny}, 9),\ \bar{\forall}\big((\mathit{flight}(x, y, w) \wedge t \geqslant w) \Rightarrow \mathit{travel}(x, y, t)\big),\ \bar{\forall}\big((\mathit{flight}(x, z, t_1) \wedge \mathit{travel}(z, y, t_2) \wedge t \geqslant t_1 + t_2) \Rightarrow \mathit{travel}(x, y, t)\big)\big\}$$

and $G \equiv \forall t(t > 1.5 \Rightarrow \exists y\ \mathit{travel}(\mathit{mad}, y, t))$.

The following is a derivation of the sequent $\Delta; \{\top\} \vdash G$. We use the abbreviations $\Gamma = \{\top, t > 1.5, y \approx \mathit{par}\}$ and $\Gamma' = \Gamma \cup \{x' \approx \mathit{mad}, y' \approx \mathit{par}, t' \approx t, w' \approx 1.5\}$. $(\exists^4)$ denotes 4 successive applications of the rule $(\exists)$, the four corresponding side conditions referring to $\vdash_{\mathcal{FR}}$ are put together, and abbreviated as $\Gamma \vdash_{\mathcal{FR}} \exists x'(x' \approx \mathit{mad}) \ldots \Gamma' \vdash_{\mathcal{FR}} \exists w'(w' \approx 1.5)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash_{\mathcal{FR}} x' \approx \mathit{mad} \wedge \cdots \wedge t' \geqslant w'}{\Delta;\Gamma' \vdash x' \approx \mathit{mad} \wedge \cdots \wedge t' \geqslant w'}\ (C) \quad \dfrac{\Gamma' \vdash_{\mathcal{FR}} x' \approx \mathit{mad} \wedge y' \approx \mathit{par} \wedge w' \approx 1.5}{\Delta;\Gamma' \vdash x' \approx \mathit{mad} \wedge y' \approx \mathit{par} \wedge w' \approx 1.5}\ (C)}{\Delta;\Gamma' \vdash (x' \approx \mathit{mad} \wedge \cdots \wedge t' \geqslant w') \wedge \mathit{flight}(x', y', w')}\ (\wedge)}{\Delta;\Gamma' \vdash \mathit{flight}(x', y', w')}\ (\mathit{Clause})}{\Delta;\{t > 1.5, y \approx \mathit{par}\} \vdash \mathit{travel}(\mathit{mad}, y, t)}\ (\mathit{Clause}) \quad \{t > 1.5\} \vdash_{\mathcal{FR}} \exists y(y \approx \mathit{par})}{\Delta;\{t > 1.5\} \vdash \exists y\ \mathit{travel}(\mathit{mad}, y, t)}\ (\exists)}{\Delta;\{\top\} \vdash t > 1.5 \Rightarrow \exists y\ \mathit{travel}(\mathit{mad}, y, t)}\ (\Rightarrow_C)}{\Delta;\{\top\} \vdash \forall t\ (t > 1.5 \Rightarrow \exists y\ \mathit{travel}(\mathit{mad}, y, t))}\ (\forall)$$

$$\frac{\Gamma \vdash_{\mathcal{FR}} \exists x'(x' \approx \mathit{mad}) \ldots \Gamma' \vdash_{\mathcal{FR}} \exists w'(w' \approx 1.5)}{\Delta;\Gamma \vdash \exists x' \exists y' \exists t' \exists w'(x' \approx \mathit{mad} \wedge y' \approx y \wedge t \approx t' \wedge t' \geqslant w' \wedge \mathit{flight}(x', y', w'))}\ (\exists^4)$$

□

Example 8. Recall Examples 3 and 4. Let Δ be the set:

$$\big\{\bar{\forall}\big(x \geqslant x_1 \wedge x \leqslant x_2 \wedge y \geqslant y_1 \wedge y \leqslant y_2 \Rightarrow \mathit{rectangle}(x_1, y_1, x_2, y_2, x, y)\big)\big\},$$

and $G \equiv \mathit{rectangle}(0, 0, 4, 4, x, y), \neg \mathit{rectangle}(1, 1, 3, 3, x, y)$. The answer constraint:

$$\begin{aligned} C \equiv\ & \big((y > 3) \wedge (y \leqslant 4) \wedge (x \geqslant 0) \wedge (x \leqslant 4)\big) \vee \\ & \big((y \geqslant 0) \wedge (y < 1) \wedge (x \geqslant 0) \wedge (x \leqslant 4)\big) \vee \\ & \big((y \geqslant 0) \wedge (y \leqslant 4) \wedge (x > 3) \wedge (x \leqslant 4)\big) \vee \\ & \big((y \geqslant 0) \wedge (y \leqslant 4) \wedge (x \geqslant 0) \wedge (x < 1)\big) \end{aligned}$$

can be obtained by the following deduction:

$$\dfrac{\dfrac{\dfrac{C \vdash_{\mathcal{R}} \exists a_1 \exists a_2 \exists b_1 \exists b_2 \exists x_1 \exists y_1 (a_1 \approx 0 \wedge x_1 \approx x \wedge \cdots)}{\Delta; C \vdash \exists a_1 \exists a_2 \exists b_1 \exists b_2 \exists x_1 \exists y_1 (a_1 \approx 0 \wedge x_1 \approx x \wedge x_1 \geqslant a_1 \wedge \\ a_2 \approx 0 \wedge y_1 \approx y \wedge x_1 \leqslant b_1 \wedge b_1 \approx 4 \wedge y_1 \geqslant a_2 \wedge b_2 \approx 4 \wedge y_1 \leqslant b_2)}\ (C)}{\Delta; C \vdash \mathit{rectangle}(0, 0, 4, 4, x, y)}\ (\mathit{Clause}) \quad \mathbf{D}}{\Delta; C \vdash \mathit{rectangle}(0, 0, 4, 4, x, y) \wedge \neg \mathit{rectangle}(1, 1, 3, 3, x, y)}\ (\wedge)$$

where **D** is a deduction for $\Delta; C \vdash \neg \mathit{rectangle}(1, 1, 3, 3, x, y)$ whose last steps have the form:

$$\dfrac{C \vdash_{\mathcal{R}} \neg(\ x \geqslant 1 \wedge y \geqslant 1 \wedge \\ x \leqslant 3 \wedge y \leqslant 3) \quad \dfrac{\langle \mathit{rest\ of\ derivation} \rangle}{\Delta;\ x \geqslant 1 \wedge y \geqslant 1 \wedge \\ x \leqslant 3 \wedge y \leqslant 3\ \vdash \mathit{rectangle}(1, 1, 3, 3, x, y)}}{\Delta; C \vdash \neg \mathit{rectangle}(1, 1, 3, 3, x, y)}\ (\neg) \qquad \square$$

In order to define a sound and complete goal solving procedure, some finiteness conditions must be imposed to make viable the metarule (¬). That is, it has to be guaranteed that the set of answer constraints for an atom (that occurs negated in a goal) is finite, and that this set can be computed in a finite number of steps. As usual in the constraint database field, finiteness of the set of computed answers can be ensured by imposing different safety conditions to the constraint systems [42]. A technique that guarantees termination, provided finiteness of the constraint answers sets, is stratification.

We have adopted it because it is easy to combine with our notion of constraint system, giving a clear operational semantics to the scheme $HH_{\neg}(\mathcal{C})$ by providing meaning to the whole database (in the presence of safety conditions).

The stratified negation that we propose is widely explained in the next section, where a stratified fixpoint semantics is presented as the basis of an implementation of $HH_{\neg}(\mathcal{C})$.

5. Fixpoint semantics

We have extended and adapted the semantics presented in [19] in order to interpret full $HH_{\neg}(\mathcal{C})$ using a stratification technique. The semantics defined was based on a *forcing relation* among programs, sets of constraints and goals that states whether an interpretation makes true a goal G in the context $\langle \Delta, \Gamma \rangle$ of a program Δ and a set of constraints Γ. Interpretations were defined as functions able to give meaning to every pair $\langle \Delta, \Gamma \rangle$ as sets of atoms. The interpretation had to depend on this context because, when computing implicative goals, Δ or Γ may be augmented. Here, interpretations are defined as functions able to give meaning to a database as a set of pairs (*Atom*, *Constraint*), and are classified on strata. Following [51], the stratification of a database is based on the definition of a dependency graph. Next we introduce these notions for our language.

5.1. Stratification and dependency graph

Given a set of clauses and goals Φ, the corresponding dependency graph DG_{Φ} is a directed graph whose nodes are the defined predicate symbols in Φ, and the edges are determined by the implication symbols of the formulas.

Here, we adapt those notions as a useful starting point of a fixpoint semantics for our language. But now, the construction of dependency graphs must consider the fact that implications may occur not only between the head and the body of a clause, but also inside the goals, and therefore in any clause body. This feature will be taken into account in the following way: An implication of the form $F_1 \Rightarrow F_2$ produces edges in the graph from the defined predicate symbols inside F_1 to every defined predicate symbol inside F_2. An edge will be negatively labeled when the corresponding atom occurs negated on the left of the implication. Since constraints do not include defined predicate symbols, they cannot produce dependencies.

Example 9. Let Δ be the bank database of Example 6. Fig. 5 shows the dependency graph for Δ (the predicate *branch* corresponds to an isolated node which is not represented in the figure). Negative edges are labeled with ¬. □

Definition 1. Given a set of formulas Φ, its corresponding dependency graph DG_{Φ}, and two predicates p and q, we say:

- q *depends* on p if there is a path from p to q in DG_{Φ}.
- q *negatively depends* on p if there is a path from p to q in DG_{Φ} with at least one negatively labeled edge.

Definition 2. Let Φ be a set of formulas and $P = \{p_1, \ldots, p_n\}$ the set of defined predicate symbols of Φ. A *stratification* of Φ is any mapping $s : P \to \{1, \ldots, n\}$ such that $s(p) \leqslant s(q)$ if q depends on p, and $s(p) < s(q)$ if q negatively depends on p. Φ is stratifiable if there is a stratification for it.

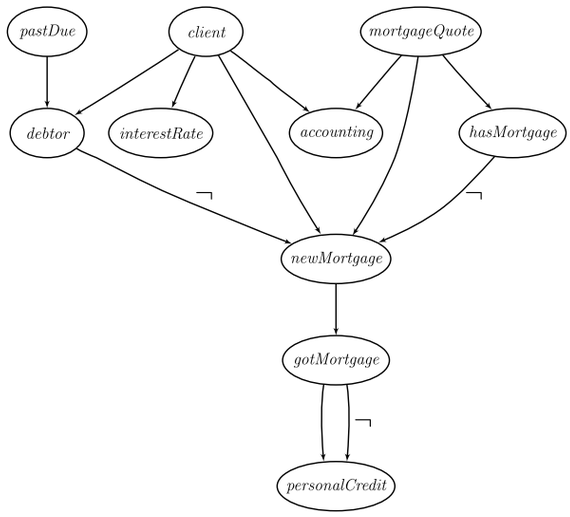


Fig. 5. Dependency graph for Example 6.

Example 10. A stratification for the database Δ of Example 5 will collect all the predicates in the stratum 1 except *nondeltravel* and *trip*, which will be in stratum 2. Intuitively, this means that for evaluating *nondeltravel*, the rest of predicates (except *trip*) should be evaluated before (in particular, *delayed*). Formulating the query: $G \equiv \exists t\ \mathit{deltravel}(x, y, t) \Rightarrow \mathit{delayed}(x, y)$, the augmented set $\Delta \cup \{G\}$ remains stratifiable, but if $G' \equiv \mathit{trip}(\mathit{mad}, \mathit{lon}, T) \Rightarrow \mathit{delay}(\mathit{mad}, \mathit{ny}, t)$ is formulated, the extended set $\Delta \cup \{G'\}$ results non-stratifiable. It is because G' adds the dependency $\mathit{trip} \rightarrow \mathit{delay}$, and then, any stratification s must satisfy $s(\mathit{trip}) \leqslant s(\mathit{delay}) \leqslant s(\mathit{delayed}) < s(\mathit{nondeltravel}) \leqslant s(\mathit{trip})$, which is impossible. □

From now on, we assume the existence of a fixed stratification s for the considered sets $\Delta \cup \{G\}$. It is useful to have a notion of the stratum of an atom (i.e., the stratum of its predicate symbol), but also to extend this notion to any formula or set of formulas.

Definition 3. Let F be a goal or a clause. The *stratum of a formula* F, denoted by $\mathit{str}(F)$, is recursively defined as:

$$\begin{aligned}
&\mathit{str}(C) = 1 \\
&\mathit{str}(\neg A) = 1 + \mathit{str}(A) && \mathit{str}\big(p(t_1, \ldots, t_n)\big) = s(p) \\
&\mathit{str}(F_1 \square F_2) = \mathit{max}\big(\mathit{str}(F_1), \mathit{str}(F_2)\big), && \text{where } \square \in \{\wedge, \vee, \Rightarrow\} \\
&\mathit{str}(Q\,xF) = \mathit{str}(F), && \text{where } Q \in \{\exists, \forall\}
\end{aligned}$$

The *stratum of a set of formulas* Φ is $\mathit{str}(\Phi) = \mathit{max}\{\mathit{str}(F) \mid F \in \Phi\}$.

5.2. Stratified interpretations and forcing relation

Let $\mathcal{W}$ be the set of stratifiable databases with respect to the same fixed stratification s, which can be built from a particular signature. Interpretations and the fixpoint operator will be applied to the databases of $\mathcal{W}$ and they operate over strata.

Let At be the set of open atoms, i.e., defined predicate symbols of the signature applied to variables (up to variable renaming); and let $\mathcal{SL}_{\mathcal{C}}$ be the set of $\mathcal{C}$-satisfiable constraints modulo $\mathcal{C}$-equivalence. The set $At \times \mathcal{SL}_{\mathcal{C}}$ is finite because we consider finite signatures and compact constraint systems. As we define below, an interpretation over a stratum i of a database will be a set of pairs $(A, [C]) \in At \times \mathcal{SL}_{\mathcal{C}}$, where $\mathit{str}(A) \leqslant i$ and $[C]$ represents the set of constraints $\mathcal{C}$-equivalent to C.

Definition 4. Let $i \geqslant 1$. An *interpretation I over the stratum i* is a function $I : \mathcal{W} \rightarrow \mathcal{P}(At \times \mathcal{SL}_{\mathcal{C}})$, such that for every $\Delta \in \mathcal{W}$, if $(A, [C]) \in I(\Delta)$ then $\mathit{str}(A) \leqslant j$. We denote by $\mathcal{I}_i$ the *set of interpretations over* i.

In order to simplify the notation, we write:

- $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, instead of $(A, [C])$, assuming that C is any representative of its equivalence class $[C]$.
- $[I(\Delta)]_i$ to represent the set $\{(A, C) \in I(\Delta) \mid str(A) = i\}$.

Notice that if $str(\Delta) = k$, then $\{[I(\Delta)]_i \mid 1 \leqslant i \leqslant k\}$ is a partition of $I(\Delta)$. For every $i \geqslant 1$, an order on $\mathcal{I}_i$ can be defined as follows.

Definition 5. Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$. I_1 *is less than or equal to* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if for each $\Delta \in \mathcal{W}$ the following conditions are satisfied:

- $[I_1(\Delta)]_j = [I_2(\Delta)]_j$, for every $1 \leqslant j < i$.
- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$.

It is straightforward to check that for any $i \geqslant 1$, $(\mathcal{I}_i, \sqsubseteq_i)$ is a poset. The idea behind this definition is that when an interpretation over a stratum i increases, the information of the smaller strata remains invariable. In such a way, if $str(\neg A) = i$, since $str(A) = i - 1$, the truth value of $\neg A$ at the stratum i will remain invariable and monotonicity of the truth relation can be guaranteed even for negative atoms, as we will show. In addition, the following result holds.

Lemma 1. *For any* $i \geqslant 1$, *any chain of interpretations of* $(\mathcal{I}_i, \sqsubseteq_i)$, $\{I_n\}_{n \geqslant 0}$, *such that* $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$, *has a least upper bound* $\bigsqcup_{n \geqslant 0} I_n$, *which can be defined as*: $(\bigsqcup_{n \geqslant 0} I_n)(\Delta) = \bigcup\{I_n(\Delta) \mid n \geqslant 0\}$, *for any* $\Delta \in \mathcal{W}$.

Proof. For any $\Delta \in \mathcal{W}$ we define the function $\hat{I}$ as $\hat{I}(\Delta) = \bigcup\{I_n(\Delta) \mid n \geqslant 0\}$, or simply $\bigcup_{n \geqslant 0} I_n(\Delta)$. It must be checked that $\hat{I}$ is the least upper bound of the chain $\{I_n\}_{n \geqslant 0}$.

- $\hat{I}$ is an upper bound of the chain. Let $k \geqslant 0$. For any Δ, we have that:
  $[I_k(\Delta)]_j = \bigcup_{n \geqslant 0}[I_n(\Delta)]_j = [\hat{I}(\Delta)]_j$, for $1 \leqslant j < i$, and
  $[I_k(\Delta)]_i \subseteq \bigcup_{n \geqslant 0}[I_n(\Delta)]_i = [\hat{I}(\Delta)]_i$.
- Now, we prove that it is the least upper bound of the chain.
  Let us assume that I' is an upper bound of $\{I_n\}_{n \geqslant 0}$. For each $k \geqslant 0$, $I_k \sqsubseteq_i I'$ implies that for any Δ, $[I_k(\Delta)]_j = [I'(\Delta)]_j$, for $1 \leqslant j < i$, and $[I_k(\Delta)]_i \subseteq [I'(\Delta)]_i$. Therefore, $\bigcup_{n \geqslant 0}[I_n(\Delta)]_j = [I'(\Delta)]_j$, for $1 \leqslant j < i$, and $\bigcup_{n \geqslant 0}[I_n(\Delta)]_i \subseteq [I'(\Delta)]_i$, for any $\Delta \in \mathcal{W}$. Thus, $\hat{I} \sqsubseteq_i I'$. □

The following definition formalizes the notion of a query G being "true" for an interpretation I in the context of a database Δ, if the constraint C is satisfied.

As already said, we assume that s is not only a stratification for Δ, but also for $\Delta \cup \{G\}$.

Definition 6. Let $i \geqslant 1$. The *forcing relation* $\models$ between pairs I, Δ and pairs (G, C) (where $I \in \mathcal{I}_i$, $str(G) \leqslant i$, and C is $\mathcal{C}$-satisfiable) is recursively defined by the rules below. When $I, \Delta \models (G, C)$, it is said that (G, C) is *forced* by I, Δ.

- $I, \Delta \models (C', C) \iff C \vdash_{\mathcal{C}} C'$.
- $I, \Delta \models (A, C) \iff (A, C) \in I(\Delta)$.
- $I, \Delta \models (\neg A, C) \iff$ for every $(A, C') \in I(\Delta)$, $C \vdash_{\mathcal{C}} \neg C'$ holds, and if there is no pair of the form (A, C') in $I(\Delta)$, then $C \equiv \top$.
- $I, \Delta \models (G_1 \wedge G_2, C) \iff$ for each $i \in \{1, 2\}$, $I, \Delta \models (G_i, C)$.
- $I, \Delta \models (G_1 \vee G_2, C) \iff$ for some $i \in \{1, 2\}$ $I, \Delta \models (G_i, C)$.
- $I, \Delta \models (D \Rightarrow G, C) \iff I, \Delta \cup \{D\} \models (G, C)$.
- $I, \Delta \models (C' \Rightarrow G, C) \iff I, \Delta \models (G, C \wedge C')$.
- $I, \Delta \models (\exists x G, C) \iff$ there is C' such that $I, \Delta \models (G[y/x], C')$, where y does not occur free in Δ, $\exists x G$, C, and $C \vdash_{\mathcal{C}} \exists y C'$.
- $I, \Delta \models (\forall x G, C) \iff I, \Delta \models (G[y/x], C)$ where y does not occur free in Δ, $\forall x G$, C.

Those rules are well-defined because if s is a stratification for $\Delta \cup \{G\}$, with $str(G) \leqslant i$, and G' is a subformula of G, then s is also a stratification for $\Delta \cup \{G'\}$, and $str(G') \leqslant i$. Notice that, for the particular case $G \equiv D \Rightarrow G'$, s will be also a stratification for $\Delta \cup \{D, G'\}$.

From now on, when we write $I, \Delta \models (G, C)$ we will assume that if $I \in \mathcal{I}_i$, then $str(G) \leqslant i$ and C is $\mathcal{C}$-satisfiable. The relation $\models$ is not defined otherwise. Formally, $\models$ should be denoted $\models_i$, because there is a forcing relation for each $\mathcal{I}_i$. We avoid the subindex in order to simplify the notation.

The following lemma establishes the monotonicity of the forcing relation.

Lemma 2. *Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$ such that $I_1 \sqsubseteq_i I_2$. Then, for any $\Delta \in \mathcal{W}$, and $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, if $I_1, \Delta \models (G, C)$, then $I_2, \Delta \models (G, C)$.*

Proof. The proof is inductive on the structure of G. The full proof is in Appendix A. Here only a few significative cases are presented.

$(\neg A)$ If $I_1, \Delta \models (\neg A, C)$, then $C \vdash_{\mathcal{C}} \neg C'$ for every C' such that $(A, C') \in I_1(\Delta)$, or there is no such C' and $C \equiv \top$. Since $str(\neg A) \leqslant i$, obviously $str(A) = j$, for some $j < i$. Then $[I_2(\Delta)]_j = [I_1(\Delta)]_j$, because $I_1 \sqsubseteq_i I_2$, and $I_1, \Delta \models (\neg A, C)$ is equivalent to $I_2, \Delta \models (\neg A, C)$.

$(\forall x G')$ $I_1, \Delta \models (\forall x G', C) \iff \Delta \models (G'[y/x], C)$, where y does not occur free in Δ, $\forall x G'$, C. By induction hypothesis $I_2, \Delta \models (G'[y/x], C)$, therefore $I_2, \Delta \models (\forall x G', C)$. □

Lemma 3. *Let $i \geqslant 1$ and let $\{I_n\}_{n \geqslant 0}$ be a denumerable family of interpretations over the stratum i, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$. Then, for any Δ, G and C,*

$$\bigsqcup_{n \geqslant 0} I_n, \Delta \models (G, C) \iff \text{there exists } k \geqslant 0 \text{ such that } I_k, \Delta \models (G, C).$$

Proof. In order to simplify the notation we write $\hat{I}$ for $\bigsqcup_{n \geqslant 0} I_n$. The implication from right to left is a consequence of Lemma 2, since $I_k \sqsubseteq_i \hat{I}$ holds for any k. The converse is proved using the result of Lemma 1 ($\hat{I}(\Delta) = \bigcup_{n \geqslant 0} I_n(\Delta)$), by induction on the structure of G. As before, we present only some cases, and the others appear in Appendix A.

$(\neg A)$ $\hat{I}, \Delta \models (\neg A, C) \iff$ for every C' such that $\hat{I}, \Delta \models (A, C')$, $C \vdash_{\mathcal{C}} \neg C'$, or there is not such C'. We are assuming that $str(\neg A) \leqslant i$ so $str(A) < i$. $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$ implies that $[I_0(\Delta)]_j = [I_1(\Delta)]_j = \cdots = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j$. So for any $k \geqslant 1$, $I_k, \Delta \models (\neg A, C)$.

$(D \Rightarrow G')$ $\hat{I}, \Delta \models (D \Rightarrow G', C) \iff \hat{I}, \Delta \cup \{D\} \models (G', C) \Rightarrow$ there is $k \geqslant 0$ such that $I_k, \Delta \cup \{D\} \models (G', C)$, by induction hypothesis $\Rightarrow$ there is $k \geqslant 0$ such that $I_k, \Delta \models (D \Rightarrow G', C)$. □

Next, a continuous operator for every stratum transforming interpretations is defined. Its least fixpoint supplies the expected version of truth at each stratum.

Definition 7. Let $i \geqslant 1$ represent a stratum. The *operator* $T_i : \mathcal{I}_i \to \mathcal{I}_i$ transforms interpretations over i as follows. For any $I \in \mathcal{I}_i$, $\Delta \in \mathcal{W}$, and $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, $(A, C) \in (T_i(I))(\Delta)$ when:

- $(A, C) \in [I(\Delta)]_j$ for some $j < i$ or
- $str(A) = i$ and there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause in $elab(\Delta)$, such that the variables $\bar{x}$ do not occur free in A, and $I, \Delta \models (\exists \bar{x}(A \approx A' \wedge G), C)$.

The crucial aspect of T_i is: For a database Δ, T_i incorporates information obtained exclusively from the clauses of Δ, whose heads are atoms of the stratum i, and the information of smaller strata remains invariable. Notice that if $str(A) = i$, then $str(\exists \bar{x}(A \approx A' \wedge G)) \leqslant i$ and T_i is well-defined.

In order to establish the existence of a fixpoint of T_i, it will be proved to be monotonous and continuous.

Lemma 4 *(Monotonicity of T_i). Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$ such that $I_1 \sqsubseteq_i I_2$. Then, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.*

Proof. Let us consider any Δ and $(A, C) \in (T_i(I_1))(\Delta)$. This implies that $str(A) \leqslant i$. If $str(A) = j < i$, then $(A, C) \in [I_1(\Delta)]_j = [I_2(\Delta)]_j$, because $I_1 \sqsubseteq_i I_2$ and $j < i$. Hence $(A, C) \in (T_i(I_2))(\Delta)$, by definition of T_i. If $str(A) = i$, then there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ, such that the variables $\bar{x}$ do not occur free in A, and $I_1, \Delta \models (\exists \bar{x}(A \approx A' \wedge G), C)$. Using Lemma 2 and the fact that $I_1 \sqsubseteq_i I_2$, we obtain $I_2, \Delta \models (\exists \bar{x}(A \approx A' \wedge G), C)$, which implies $(A, C) \in T_i(I_2)(\Delta)$, by definition of T_i. □

Lemma 5 *(Continuity of T_i). Let $i \geqslant 1$ and $\{I_n\}_{n \geqslant 0}$ be a denumerable family of interpretations over i, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$. Then $T_i(\bigsqcup_{n \geqslant 0} I_n) = \bigsqcup_{n \geqslant 0} T_i(I_n)$.*

Proof. The inclusion $\supseteq$ is a consequence of the monotonicity of T_i. Let us prove the inclusion $\subseteq$. Consider any Δ and $(A, C) \in (T_i(\bigsqcup_{n \geqslant 0} I_n))(\Delta)$. Then $str(A) \leqslant i$. If $str(A) = j < i$, $(A, C) \in [(T_i(\bigsqcup_{n \geqslant 0} I_n))(\Delta)]_j = [I_0(\Delta)]_j$, then $(A, C) \in (T_i(I_0))(\Delta) \subseteq \bigcup_{n \geqslant 0}(T_i(I_n))(\Delta) = (\bigsqcup_{n \geqslant 0} T_i(I_n))(\Delta)$. If $str(A) = i$, there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ, such that the variables $\bar{x}$ do not occur free in A, and $\bigsqcup_{n \geqslant 0} I_n, \Delta \models (\exists \bar{x}\ (A \approx A' \wedge G), C)$. Thanks to Lemma 3, there exists

Stratum	Iteration	Considered clause	Deduced pair	
1	$T_1(\emptyset)$	$p(a)$	$(p(x), x \approx a)$	fix_1
		$p(b)$	$(p(x), x \approx b)$	
		$\forall x(p(x) \Rightarrow t(x))$	None $(*)$	
	$T_1(T_1(\emptyset))$	$p(a)$	None $(**)$	fix_2
		$p(b)$	None $(**)$	
		$\forall x(p(x) \Rightarrow t(x))$	$(t(x), x \approx a \vee x \approx b)$	
2	$T_2(fix_1)$	$\forall x(\neg p(x) \Rightarrow q(x))$	$(q(x), x \not\approx a \wedge x \not\approx b)$	fix_3
		$\forall x((p(x) \Rightarrow q(x)) \Rightarrow r(x))$	None $(*)$	
3	$T_3(fix_2)$	$\forall x(\neg q(x) \Rightarrow u(x))$	$(u(x), x \approx a \vee x \approx b)$	

Fig. 6. Stratified fixpoint of the elaborated database of Example 11.

$k \geqslant 0$, such that $I_k, \Delta \models (\exists \bar{x}(A \approx A' \wedge G), C)$, and therefore $(A, C) \in (T_i(I_k))(\Delta)$. As a consequence, $(T_i(\bigsqcup_{n\geqslant 0} I_n))(\Delta) \subseteq \bigcup_{n\geqslant 0}(T_i(I_n))(\Delta) = (\bigsqcup_{n\geqslant 0} T_i(I_n))(\Delta)$. □

Proposition 1. *The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geqslant 0} T_1^n(I_\perp)$, where the interpretation $I_\perp$ represents the constant function $\emptyset$.*

Proof. By the Knaster–Tarski fixpoint theorem [49], using Lemma 5. □

Let fix_1 denote $\bigsqcup_{n\geqslant 0} T_1^n(I_\perp)$, i.e., the least fixpoint at stratum 1.

Consider now the following sequence $\{T_2^n(fix_1)\}_{n\geqslant 0}$ of interpretations in $(\mathcal{I}_2, \sqsubseteq_2)$. Using the properties of T_i, it is easy to prove by induction on $n \geqslant 0$ that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2\big(T_2(fix_1)\big) \sqsubseteq_2 \cdots \sqsubseteq_2 T_2^n(fix_1) \sqsubseteq_2 \cdots.$$

As before, in accordance to Lemmas 1 and 5, $\{T_2^n(fix_1)\}_{n\geqslant 0}$ has a least upper bound, $\bigsqcup_{n\geqslant 0} T_2^n(fix_1)$, in $(\mathcal{I}_2, \sqsubseteq_2)$ that is a fixpoint of T_2, denoted by fix_2. Proceeding successively in the same way, a chain:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i\big(T_i(fix_{i-1})\big) \sqsubseteq_i \cdots \sqsubseteq_i T_i^n(fix_{i-1}) \sqsubseteq_i \cdots$$

can be defined for any stratum $i > 1$, and a fixpoint of it

$$fix_i = \bigsqcup_{n\geqslant 0} T_i^n(fix_{i-1})$$

can be found. In particular, if $str(\Delta) = k$, we simplify fix_k writing fix. Then, $fix(\Delta)$ represents the pairs (A, C) such that A can be deduced from Δ if C is satisfied. Notice that $fix(\Delta)$ is computed by saturating strata sequentially from $fix_1(\Delta)$ up to $fix_k(\Delta)$, using for every i only the clauses of the stratum i.

Example 11. Given the finite domain $[a, b, c]$, let us consider the elaborated database:

$$\Delta = \big\{p(a), p(b), \forall x\big(p(x) \Rightarrow t(x)\big), \forall x\big(\neg p(x) \Rightarrow q(x)\big), \forall x\big(\big(p(x) \Rightarrow q(x)\big) \Rightarrow r(x)\big), \forall x\big(\neg q(x) \Rightarrow u(x)\big)\big\}.$$

In this database, p and t belong to stratum 1, q and r belong to stratum 2 and u belongs to stratum 3. In Fig. 6 we summarize the pairs that compose the successive iterations of the corresponding fixpoint for each stratum. Note that there are two situations in which the considered clause does not lead to new pairs in the current interpretation, if they are already in it $(**)$, and if it is not possible to find any constraint allowing to force the body of the clause $(*)$. The forcing relation is non-deterministic, therefore the constraint we show for each pair corresponds to a representative of its equivalence class. □

5.3. Soundness and completeness

The fixpoint semantics defined in [19] for $HH(\mathcal{C})$ was proven to be sound and complete with respect to the calculus $\mathcal{UC}$. Now, the coexistence of constraints, negation and implication, as well as the use of $HH_\neg(\mathcal{C})$ as a database system have made necessary to extend $\mathcal{UC}$ to $\mathcal{UC}_\neg$ and to define the concept of stratified interpretation composed of pairs (*Atom, Constraint*), instead of simply atoms as it was done in [19]. The new fixpoint semantics combines stratification techniques with the forcing relation. In this section we prove soundness and completeness of the new fixpoint semantics for $HH_\neg(\mathcal{C})$ with respect to the extended calculus $\mathcal{UC}_\neg$. This means that the forcing relation, considering the least fixpoint at the last stratum of a database and a query, coincides with derivability in $\mathcal{UC}_\neg$. More precisely, if $str(G) = i$, then (G, C) is forced by fix_i in the context of Δ if and only if C is an answer constraint of G from Δ. Although without negation, any database Δ

and query G have a stratification with only one stratum, and then soundness and completeness are similar to those results for $HH(\mathcal{C})$, the general case is not trivial. For this reason, we present the proof of the soundness and completeness in rather detail.

The definitions below introduce a measure of complexity which will be used to perform induction in the proof of soundness. Let $i \geqslant 1$ and $\mathcal{S}_i = \{\langle \Delta, G, C\rangle \in \mathcal{W} \times \mathcal{G} \times \mathcal{SL}_{\mathcal{C}} \mid \mathit{fix}_i, \Delta \models (G, C)\}$. Given any $\langle \Delta, G, C\rangle \in \mathcal{S}_i$, Lemma 3 guarantees that the set $\mathcal{S} = \{k \geqslant 0 \mid T_i^k(\mathit{fix}_{i-1}), \Delta \models (G, C)\}$[1] is non-empty. Therefore, it is possible to define $\mathit{ord}(\langle \Delta, G, C\rangle) = \min \mathcal{S}$. Let us consider the partially ordered set $(\mathcal{S}_i, <_i)$, where $<_i$ is defined as follows. Given any $\langle \Delta, G, C\rangle$, $\langle \Delta', G', C'\rangle \in \mathcal{S}_i$, $\langle \Delta, G, C\rangle <_i \langle \Delta', G', C'\rangle$ if:

- $\mathit{ord}(\langle \Delta, G, C\rangle) < \mathit{ord}(\langle \Delta', G', C'\rangle)$, or
- $\mathit{ord}(\langle \Delta, G, C\rangle) = \mathit{ord}(\langle \Delta', G', C'\rangle)$ and G is a renaming of a strict subformula of G'.

Such partial order is well-founded.

Proposition 2. *For every $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, such that $\mathit{str}(G) = 1$ then:*

$$\mathit{fix}_1, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

Proof. The proof is an adaptation of those presented in [19] to the definition of the forcing relation defined now for $HH_\neg(\mathcal{C})$. Notice that, since we are assuming that $\mathit{str}(G) = 1$, then the case $G \equiv \neg A$ has not to be considered. □

Now, we deal with the general case.

Proposition 3. *For every $i \geqslant 1$, $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, such that G does not contain negation, if $\mathit{str}(G) \leqslant i$, then:*

$$\mathit{fix}_i, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

Proof. The more difficult case to prove is for G atomic. We show here the proof of this case. The full proof is detailed in Appendix A.

$\Rightarrow$) This implication can be proved by induction on the structural order $(\mathcal{S}_i, <_i)$. Let us take $\langle \Delta, G, C\rangle \in \mathcal{S}_i$ and assume that, for any other $\langle \Delta', G', C'\rangle \in \mathcal{S}_i$, $\langle \Delta', G', C'\rangle <_i \langle \Delta, G, C\rangle$ implies that $\Delta'; C' \vdash_{\mathcal{UC}_\neg} G'$. Then, let us conclude $\Delta; C \vdash_{\mathcal{UC}_\neg} G$ by case analysis on the structure of G. For the case $G \equiv A$:

$\langle \Delta, A, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, \Delta \models (A, C)$. Let $k = \mathit{ord}(\langle \Delta, A, C\rangle)$, then $T_i^k(\mathit{fix}_{i-1}), \Delta \models (A, C)$, which is equivalent to $(A, C) \in (T_i^k(\mathit{fix}_{i-1}))(\Delta)$. Hence, there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that the variables $\bar{x}$ do not occur free in A, and $T_i^{k-1}(\mathit{fix}_{i-1}), \Delta \models (\exists \bar{x}(A \approx A' \wedge G), C)$. In this case, $\langle \Delta, \exists \bar{x}(A \approx A' \wedge G), C\rangle <_i \langle \Delta, A, C\rangle$, so the induction hypothesis can be applied, obtaining that $\Delta; C \vdash_{\mathcal{UC}_\neg} \exists \bar{x}(A \approx A' \wedge G)$. Using the rule (*Clause*) with the elaborated clause $\forall \bar{x}(G \Rightarrow A')$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} A$.

$\Leftarrow$) It is proved by induction on the height h of the tree proof for $\Delta; C \vdash_{\mathcal{UC}_\neg} G$. The case $G \equiv A$ is an inductive case. We suppose that $\Delta; C \vdash A$ has a proof of height h. Let us prove that $\mathit{fix}_i, \Delta \models (A, C)$. Obviously, the rule employed in the bottom of such proof is (*Clause*). So there must exist a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that $\bar{x}$ do not occur free in A, and that $\Delta; C \vdash \exists \bar{x}(A \approx A' \wedge G)$ has a proof of height $h - 1$. By induction hypothesis, $\mathit{fix}_i, \Delta \models (\exists \bar{x}(A \approx A' \wedge G), C)$. Using the definition of the operator T_i, the latter implies $(A, C) \in (T_i(\mathit{fix}_i))(\Delta) = \mathit{fix}_i(\Delta)$, then $\mathit{fix}_i, \Delta \models (A, C)$. □

Termination is guaranteed by the previous lemmas if Δ is stratifiable.

Theorem 1 *(Soundness and completeness). For every $i \geqslant 1$, $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, if $\mathit{str}(G) \leqslant i$ then:*

$$\mathit{fix}_i, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

Proof. By induction on i. Proposition 2 is the proof of the case $i = 1$.

For $i > 1$, assume the induction hypothesis: for every Δ, G, C, with $\mathit{str}(G) \leqslant i - 1$: $\mathit{fix}_{i-1}, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G$. Proposition 3 corresponds to the proof of every case of G except for $\neg A$. Let us analyze this case: $\mathit{fix}_i, \Delta \models (\neg A, C) \iff$ for every C' such that $(A, C') \in \mathit{fix}_i(\Delta)$, it holds $C \vdash_{\mathcal{C}} \neg C'$, or there is no such C' and $C \equiv \top$. Obviously, $\mathit{str}(A) \leqslant i - 1$, then the previous sentence is equivalent to say that for every C' such that $\mathit{fix}_{i-i}, \Delta \models (A, C')$, it holds $C \vdash_{\mathcal{C}} \neg C'$, or there is no such C' and $C \equiv \top$. Applying the induction hypothesis, it is equivalent to say that either for every C' such that $\Delta; C' \vdash_{\mathcal{UC}_\neg} A$, $C \vdash_{\mathcal{C}} \neg C'$ holds, or there is not such C' and $C \equiv \top$. This is equivalent to $\Delta; C \vdash_{\mathcal{UC}_\neg} \neg A$. □

[1] $I_\perp$ instead of fix_{i-1} for $i = 1$.

As a consequence of this theorem: $(A, C) \in \mathit{fix}(\Delta) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} A$. This means that the atoms in the fixpoint of a database are those that can be derived by the calculus.

One of the advantages of the fixpoint semantics over the proof-theoretic one is that the former provides a model for the whole database, while the latter focuses only on the meaning of a query in the context of a database. In [31] a goal solving procedure for $HH(\mathcal{C})$ based on the uniform calculus (without negation) was presented, but the presence of the negation in the language makes unavailing that procedure. The fixpoint semantics can be considered as the formal basis of particular implementations of database systems based on $HH_\neg(\mathcal{C})$. In fact, the fixpoint of a database definition will correspond to the instance of the database. Once that instance is computed, the forcing relation, in which this semantics is based on, provides a formal basis for a goal-oriented computation of answers.

Notice that the previous formalisms are defined for a generic constraint system $\mathcal{C}$ as a black box for which the existence of a solver which checks $\mathcal{C}$-satisfiability has been assumed. So, we have easily developed an implementation for the scheme $HH_\neg(\mathcal{C})$, based on this semantics, which is independent from any constraint system.

The rest of the paper is devoted to present the implementation of our language as a database system.

6. Introducing a database system based on $HH_\neg(\mathcal{C})$

Having stated $HH_\neg(\mathcal{C})$ as a formal language, this section deals with a general description of our proposal to the implementation of a Constraint Deductive Database including $HH_\neg(\mathcal{C})$ as a query language. Also, here we show some examples of use of the system.

Two main components can be distinguished in the implementation of this query language. One corresponds to the implementation of the fixpoint semantics which is very close to the theory and is independent of the concrete constraint system (cf. Section 8). The other component corresponds to the implementation of the constraint system (cf. Section 7).

The system is available at https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems including a user-oriented manual and a bundle of examples. In the following, the concrete syntax for clauses and queries is quite similar to the usual Prolog syntax, where predicate symbols and constants start with lowercase, whereas variables start with uppercase. In addition, we write `not` for negation, `=>` for implication, `ex(X,G)` representing $\exists x G$, `fa(X,G)` representing $\forall x G$, and `constr(Dom,C)` denoting a constraint C together with its domain. The $HH_\neg(\mathcal{C})$ system also requires explicit type declaration, `type(`*`predicate(dom_1, ..., dom_n)`*`)`, for predicates.

Although several solvers can be used together within the same database, they cannot be combined for the moment, i.e., constraints of different domains cannot be freely mixed to get a heterogeneous compound constraint. Predicates with arguments of different domains are restricted to those extensionally defined and are only intended for informative purposes. For instance, in the bank database of Example 6, in order to avoid the combination of domains in more complex relations (that would imply the combination of constraint systems during computation), we associate to each client a real value which represents the client identifier:

```
client_id(smith,1.0).    client_id(brown,2.0).    client_id(mcandrew,3.0).
```

The predicates introduced in Example 6 are defined in the same way, but changing the client names by real values.

6.1. Computation stages

Next, we briefly summarize the stages of computation performed by the system to calculate the fixpoint semantics of a database `Delta`:

1. Check and infer predicate types (cf. Section 7).
2. Build the dependency graph of `Delta` (cf. Appendix B).
3. Compute a stratification s for `Delta`, if there is any. Otherwise, the system throws an error message and stops (cf. Appendix B).
4. If the previous step succeeds, compute fix(`Delta`) (cf. Section 8).

The system keeps in memory the computed fixpoint fix(`Delta`), the stratification s consisting of a list of pairs ⟨`defined_predicate_symbol,stratum`⟩, and the dependency graph of `Delta`. Regarding the implementation of the dependency graph, defined in Section 5, new negatively labeled edges have been considered to deal with aggregate functions (cf. Section 7.1) and nested implication (cf. Section 8.1).

For instance, for the database of Example 6, $s =$ [⟨`client`, 1⟩, ⟨`pastDue`, 1⟩, ⟨`mortgageQuote`, 1⟩, ⟨`debtor`, 1⟩, ⟨`interestRate`, 1⟩, ⟨`hasMortgage`, 1⟩, ⟨`accounting`, 1⟩, ⟨`client_id`, 1⟩, ⟨`branch`, 1⟩, ⟨`newMortgage`, 2⟩, ⟨`gotMortgage`, 2⟩, ⟨`personalCredit`, 3⟩]. And fix(`Delta`) contains the pairs corresponding to the extensional database as `(client(3.0,5300.0,3000.0), true)`, as well as the pairs obtained from the intensional database:

- In stratum 1:

```
(debtor(1.0), true),
(interestRate(2.0,2.0), true),
(interestRate(X,Y), ((X=1.0, Y=5.0); (X=3.0, Y=5.0))),
(accounting(X,Y,Z), ((Y=400.0, Z=1500.0, X=2.0); (Y=100.0, Z=3000.0, X=3.0))),
(hasMortgage(X), (X=2.0;X=3.0))
```

- In stratum 2:

```
(newMortgage(X,Y), ((Y=<200.0, X=2.0); (Y=<1100.0, X=3.0))),
(gotMortgage(X), (X=2.0; X=3.0))
```

- In stratum 3:

```
(personalCredit(X,Y), ((Y>=6000.0, Y<20000.0, X/=2.0, X/=3.0);
                       (Y<6000.0, X=2.0); (Y<6000.0, X=3.0)))
```

6.2. Querying

When a query `G` is submitted at the prompt `HHn(C)>`, the system computes, if it exists, a new stratification s' for the set `Delta` $\cup$ {`G`}. If there is not such s', the query cannot be computed and the system stops; otherwise, the kept fixpoint, computed for `Delta` is valid to evaluate the query `G`. According to the theory, the answer constraint `C` must satisfy *fix*(`Delta`); `Delta` $\models$ (`G`, `C`). This forcing relation is implemented by means of the predicate `force(Delta,Stratification,I,G,C)`, as it will be explained in Section 8. This predicate makes use of the current stratification. Let us distinguish two cases, depending on whether the previous stratification s is equal to the new s' or not:

- If $s = s'$, then as *fix*(`Delta`) was calculated with s, the answer constraint `C` can be obtained by executing `force`(`Delta`, s, *fix*(`Delta`), `G`, `C`).
- If $s \neq s'$, it is because `G` contains some subgoal of the form `D => G'`. In this case, the dependency graph of `Delta`$\cup${`G`}, from which s' has been obtained, contains the edges of `Delta` plus the edges corresponding to the new implications introduced by `G`. Hence the new stratification s' is also a valid stratification for `Delta`, and the same fixpoint *fix*(`Delta`) would have been obtained by working with s'. Therefore, as before, the answer constraint `C` for the submitted query `G` can be computed by executing `force`(`Delta`, s', *fix*(`Delta`), `G`, `C`). The information of the stratification is needed, because solving `G` requires to solve the implication subgoals inside it. When some `D => G'` is going to be solved, in accordance with the definition of the forcing relation, `Delta` is locally augmented with `D` in order to find the answer of `G'`. Since s' has been defined taking into account those implications, it is also a stratification of `Delta` $\cup$ {`D`}, that will be used in this local computation.

In summary, in both cases the kept fixpoint *fix*(`Delta`) is the needed interpretation to find the answer, that is obtained just executing:

`force`(`Delta`, s', *fix*`(Delta),G,C`).

Following with the example of the bank, the user can ask whether every client belongs to Madrid office:

```
HHn(C)> fa(A,branch(mad,A)).
```

This query does not imply any change in the dependency graph, it can be solved using the kept fixpoint. A universal quantification over a finite domain is naturally translated into a conjunctive constraint obtained by instantiating the quantified variable with each element in the domain. Then the result is:

```
Answer: false
```

Dealing with negation, the user can ask for the clients that have no mortgage:

```
HHn(C)> not(hasMortgage(N)).
```

This query does not change the stratification and the system uses again the kept fixpoint. Every constraint `C` associated to a pair `(hasMortgage(N),C)` in the fixpoint is collected and a negative conjunction is built, which is sent to the solver. Finally, the system returns:

```
Answer: N/=3.0, N/=2.0
```

Let us show a contrived query only to illustrate a situation in which the stratification changes:

```
HHn(C)> newMortgage(N,R) => interestRate(N,R).
```

This query introduces a new dependency between `newMortgage` and `interestRate`, then the second predicate must be now in stratum 2. However, the already computed fixpoint is still valid for calculating the answer, which is:

```
Answer: (R=2.0, N=2.0); (R=5.0, N=1.0); (R=5.0, N=3.0)
```

7. Implementing constraint solving

This section focuses on the implementation of constraint solving for several particular constraint systems. We also introduce aggregates in our system, for the first time, as constraint functions belonging to concrete constraint systems.

7.1. Aggregates

Aggregate functions are useful for computing single values from a set of numerical or other-type values. For instance, common predefined functions of relational query languages are the average and the maximum of a numeric attribute. As our deductive database with constraints scheme gives a natural analogy of the relational calculus, a crucial question concerns with how to extend standard aggregation constructs from the relational model to our scheme in the context of a constraint domain. Certain proposals to solve this question have been formulated both in geometric constraint databases (see, e.g., Chapter 6 of [30]) and deductive database settings [47,55,54,39].

In attempting to add aggregates to constraint query languages, several obstacles come up. Aggregate functions take a set of values as input and return a single value, but the output of queries in those languages are constraints which represent intensional answers, not an explicit set of values. In addition, since a constraint answer can represent an infinite set, some aggregate functions, as `count`, make no sense.

We have taken advantage of certain aspects of the operational semantics of our database system in order to deal with these problems. On the one hand, the stratified fixpoint computation, designed to support negation, is a good framework to incorporate aggregates. The rationale behind this is ensuring monotonicity as, along building the fixpoint for a given stratum, the result of an aggregate function taking values in this stratum may change. For instance, the count of tuples corresponding to a relation is only known after such a relation has been completely computed. This is guaranteed if the sum is computed in a stratum above the relation stratum. This can be achieved by introducing a negative dependency as we will see later in this section. On the other hand, aggregates can be represented as functions of a constraint system, and then its computation can be relegated to the corresponding constraint solver.

As a general requirement for computing an aggregate function over a predicate p, each pair $(p(x_1,\ldots,x_n), C)$ in the current interpretation must hold that C restricts each variable $x_1,\ldots,x_n$ to a single value. Otherwise, the system is not able to compute it. Our supported aggregates include `count`, ranging over an atom, and `sum` (summatory), `avg` (average), `min` (minimum), and `max` (maximum), ranging over an atom and a program variable. Implementation details are given in Appendix C.2.1.

Example 12. For instance, the view:

```
liquid(Amount) :- constr(real,Amount=sum(client(N,B,S),B))
```

defined for Example 6, allows to compute the liquid assets as the cumulative sum of balances of all the clients, by including the aggregate `sum` in a constraint expression. The average salary can be specified by:

```
avg_salary(Average) :- constr(real,Average=
           sum(client(N1,B1,S1),B1)/count(client(N2,B2,S2)))
```

or directly with the aggregate `avg`:

```
avg_salary(Average) :- constr(real,Average=avg(client(N,B,S),S)).
```

The richness of the database language $HH_{\neg}(\mathcal{C})$ allows to calculate an aggregate function under an assumption which changes the values used in the aggregation. For instance, consider the following view:

```
view(X) :- pastDue(brown,200.0) => constr(real,X=sum(pastDue(N,A),A)).
```

It assumes that client Brown has a past due and then compute the `sum` of all the past dues in the database. Adding this clause to the current database, the query

```
HHn(C)> view(X)
```

has an answer `X=3300`. □

Since constraints can now contain aggregate functions, and aggregates include defined predicates, additional considerations must be taken into account in order to compute them. Computing aggregate functions requires that the involved predicates are completely known, i.e., the part of the fixpoint corresponding to those predicates has been fully computed. Similarly to what happens with a clause containing a negated atom in its body, if a clause, defining a predicate p, contains an aggregate over a predicate q, the computation of q must be finished when the computation of p begins. This condition is easily achieved by introducing a negative dependency from q to p, which guarantees $s(q) < s(p)$ in the stratification. For the clauses of Example 12, we get $s(\texttt{client}) < s(\texttt{liquid})$, $s(\texttt{client}) < s(\texttt{avg\_salary})$, and $s(\texttt{pastDue}) < s(\texttt{view})$. Full details on the dependency graph construction and stratification can be found in Appendix B.

7.2. Constraint systems and solvers

We have proposed three constraint systems as possible instances $\mathcal{C}$ of the scheme $HH_{\neg}(\mathcal{C})$: Boolean, Reals, and Finite Domains, representing a family of specific constraint systems ranging over denumerable sets. Enumerated types are included as well as (finite) integer numeric types. Our constraint systems include the concrete syntax for the required values, symbols, connectives, and quantifiers as follows: "`true`", "`false`", "`=`", "`,`", "`not`" and "`ex(X,C)`"; in addition, we also include "`;`", "`/=`" , "`>`", "`>=`", all of them with the usual meaning. Numeric constraints include arithmetic operators (as "`+`", "`-`", ...) and constraint functions (as "`abs`", ...). Moreover, Boolean and Finite Domain constraints admit the universal quantifier "`fa(X,C)`". The Finite Domain incorporates the particular domain constraint "`X in Range`", where `Range` is a subset of data values built with both `V1..V2`, which denotes the set of values in the closed interval between `V1` and `V2`, and `R1\/R2`, which denotes the union of ranges.

We have incorporated a simple type checking and inferrer mechanism in the system. It expects a type declaration for each predicate symbol in the database and warns about missing declarations or misleading use of predicates. Types for queries are inferred from the type information of the database. In addition to the well-known benefits of using a typed language, we use type information to know the constraint system a constraint belongs to.

We have considered the entailment relation of the classical logic with equality for every constraint system. This entailment satisfies the minimal condition imposed to constraint systems. For implementing this relation, we provide a constraint solver with a generic interface `solve(I,C,SC)` for $\texttt{C} \vdash_{\mathcal{C}} \texttt{SC}$, intended to solve a constraint `C`, i.e., to produce a *solved form* `SC`, if it is satisfiable, or $\bot$ otherwise. A solved form `SC` corresponding to a constraint `C` is a simplified, more readable form of the constraint w.r.t. `C`. A solved form is either a *simple constraint* or a disjunction of simple constraints, where a simple constraint is a constraint that does neither include disjunctions nor quantifications, nor negations. The generic interface `solve` for solving constraints is as follows:

```
solve(+Interpretation,+Constraint,-SolvedConstraint)
```

which solves `Constraint` as `SolvedConstraint` under `Interpretation`, where this interpretation is included to allow to compute aggregates.

For implementing the constraint systems, instead of starting completely from scratch, we rely on the underlying constraint solvers already available in SWI-Prolog [53,50]. In this way, we develop a solver layer which is built over these underlying (simpler) solvers which is capable of solving all the constraints in such (more complex) $\mathcal{C}$-instances. As an example to illustrate such an implementation, let's consider the $\mathcal{C}$-instance Finite Domains. Let's denote supported constraints in the underlying solver as primitive constraints. On the one hand, certain constraints of the constraint system Finite Domains can be mapped to primitive constraints. This mapping involves relating enumerated data values (non-integer in general) in such constraint system with integers in the underlying solver. Before posting to this solver, a constraint is rewritten with the mapped integer values and, after solving, solved constraints in the constraint store are rewritten back with the corresponding enumerated values. On the other hand, there are constraints in the $\mathcal{C}$-instance that the underlying solver cannot directly solve (quantifiers and disjunctions). For them we develop specific constraint solving as shown in Appendix C.3.

Constraint solving is expected to terminate since solver operations are always monotonic (constraints are added, possibly pruning the search space, but not removed) and enumeration (universal quantification) is only provided on finite domains. Completeness for finite domains is related to completeness of the underlying finite domain solver.

8. Implementing the fixpoint semantics

In this section we summarize the main ideas that guide the implementation of the core system: the fixpoint computation. This implementation is very close to the theoretical framework developed in previous sections, but there are some critical points, regarding nested implication, where we must take pragmatic decisions. We will provide a general view of such an implementation and provide relevant details for nested implications. The constraint solvers are used as black boxes with the appropriate interface predicate `solve`.

In this section we assume a stratifiable database Δ, with a stratification that has been previously computed and stored as an association list `Stratification`. The fixpoint is then computed stratum by stratum (although a stratum may require to compute the fixpoint for a previous stratum for the database locally enlarged due to nested implications, as we will explain in Section 8.1). The predicate

```
fixPointStrat(+Delta, +Stratification, +St, -Fix)
```

computes `Fix` $= fix_{\texttt{St}}$(`Delta`), using `Stratification`. Then, if `Delta` represents a database such that str(`Delta`) $= k$, this predicate gives fix_k(`Delta`) by computing previous fixpoints from stratum `St` $= 0$ to `St` $= k$.

Each fixpoint is evaluated by iterating the fixpoint operator following Definition 7, that relies on the forcing relation, implemented by means of the predicate

```
force(+Delta,+Stratification,+I,+G,-C)
```

Given `I` $= T_i^n(fix_{i-1})$(`Delta`), for some $n \geqslant 0$ and a fixed stratum $i > 0$, `force` is successful if

$$T_i^n(fix_{i-1}),\ \texttt{Delta} \models (\texttt{G}, \texttt{C}).$$

This predicate is implemented in a deterministic way, by making a case distinction on the syntax of the goal `G` to be forced (see Fig. 9 in Appendix D). But the theory still contains another source of non-determinism: the definition of $\models$ establishes conditions on a constraint C in order to satisfy `I,Delta` $\models$ `(G,C)`, and the predicate `force` must build a concrete constraint `C` in a deterministic way. In addition, each possible answer constraint for a goal must be captured in a single answer constraint (possibly) using disjunctions. The concrete Prolog code for the predicate `force` is presented and explained in Appendix D. Next we will explain the case of implication which is the farthest from theory.

8.1. The case of `D => G` in the forcing relation

Implementing `force(Delta,I,(D => G),C)` requires some special treatment. In this case, according to the definition of the relation $\models$ (see Definition 6), `Delta` is augmented with the clause `D`. At this point, the current set `I` has been computed w.r.t. to the database `Delta`. Then, if i and n are, respectively, the stratum and iteration under construction, $(A, C) \in$ `I` implies $(A, C) \in T_i^n(I')$(`Delta`), where I' is the fixpoint for the stratum $i - 1$, built from `Delta`. As stated in the theory, the next step will be to prove $T_i^n(I')$, `Delta`$\cup\{$`D`$\}$ $\models$ `(G,C)`. But the question is how to compute $T_i^n(I')$(`Delta`$\cup\{$`D`$\}$). Notice that `I` is not useful here. First, because $I(\Delta) \subseteq I(\Delta \cup \{D\})$ does not hold for every I, Δ, D. Second, because `I` has been built considering always `Delta`. In particular, the fixpoint I' has been computed for `Delta`, and then it represents fix_{i-1}(`Delta`). So nothing is known about the needed set $T_i^n(I')$(`Delta` $\cup\{$`D`$\}$).

The actual fact is that the definition of the fixpoint operator T_i is not constructive for the case of implication due to the increase of the set of clauses. To solve this obstacle, we have adopted a conservative position. Let `StG` $= max\{St \mid \langle p, St\rangle \in$ `Stratification`, p a predicate symbol in `G`$\}$. Then, the fixpoint of the stratum `StG` for `Delta`$\cup\{$`D`$\}$ is locally computed, and finally it is proved if $fix_{\texttt{StG}}$, `Delta`$\cup\{$`D`$\}$ $\models$ `(G,C)`.

This solution causes the following problem. Consider a clause in `Delta` of the form `A :- D => G`, such that $i = str$(`A`). From Definition 3, `StG` $\leqslant i$ can be deduced. During the computation of fix_i(`Delta`), the fixpoint operator takes this clause into account in order to look for a pair `(A,C)` to be added to the current `I`. Following Definition 7, `I,Delta` $\models$ $(\exists\bar{x}(\texttt{A} \approx \texttt{A}' \wedge$ `D => G`$), \texttt{C})$ must be proved, then after the existential quantifiers are eliminated, the predicates

```
force(Delta,Stratification,I,A≈A',C),   and
force(Delta,Stratification,I,(D => G),C)
```

will be executed (except variable renaming). The second `force` will call to

```
fixPointStrat(Delta1,Stratification,StG,Fix),
```

where `Delta1` $=$ `Delta` $\cup\{$`D`$\}$ (modulo elaboration and variable renaming). If `StG` $= i$, this means to build fix_i(`Delta1`), so the clause `A :- D => G` will be tried again, because the stratum of `A` is i. This gives rise to a non-terminating loop, since `force(Delta1,Stratification,I,(D => G),C)` will be executed and `Delta1` will be augmented with the elaboration of `D` once more, obtaining `Delta2`, which is again augmented with the same clause, and so on. However, if `StG` $< i$, then `Fix` $= fix_{\texttt{StG}}$(`Delta1`) can be correctly built, because `A :- D => G` is not considered in strata less that i.

The target `StG` $<$ str(`A`) would be ensured if the predicate with maximum stratum in `G` negatively depends on the predicate symbol of `A`. This is achieved by negatively labeling the edges from the defined predicate symbols of `G` to the predicate symbol of `A`. Although these new dependencies impose additional syntactical conditions on databases to be stratifiable, the implementation remains sound w.r.t. the stratified fixpoint semantics defined in Section 5.

The following example sketches the computation of the fixpoint for a predicate defined using an embedded implication.

Example 13. Let us consider the following database `Delta`:

```
q(a).            r(c).
q(b).            p(X) :- q(X) => r(X).
```

According to the previous explanations, if we consider a `Stratification` where all the predicates `p`, `q` and `r` are in stratum 1, then we would have the following infinite sequence of calls:

```
fixPointStrat(Delta, Stratification, 1, Fix)
fixPointStrat(DeltaU{q(X)}, Stratification, 1, Fix)
fixPointStrat(DeltaU{q(X)}U{q(X)}, Stratification, 1, Fix)
...
```

But, with the current definition of dependency graph, the `Stratification` raises `p` to stratum 2, while `q` and `r` are in stratum 1. For the first stratum `fixPointStrat(Delta,Stratification,1,Fix1)` gets `Fix1={(q(X),X=a), (q(X),X=b),(r(X),X=c)}`, because `p(X) :- q(X) => r(X)` is not considered here. For the second (and last) stratum we have the call

```
fixPointStrat(Delta,Stratification,2,Fix2)
```

`Fix2` will be built from `Fix1` introducing pairs related to `p`. In the first iteration of the fixpoint operator, and as we have remarked above, the clause defining `p` will require to execute

```
force(Delta,Stratification,Fix1,q(X) => r(X),C),
```

in order to calculate `C` and then introduce a pair `(p(X), C)` into `Fix1`. Two main steps can be distinguished in such execution (see Appendix D for details):

1. Extending the database `Delta` with `q(X)`, obtaining `Delta1` and locally evaluating the fixpoint of the stratum 1 (the stratum of `r`) for the extended database `Delta1`. This is achieved by calling

   ```
   fixPointStrat(Delta1,Stratification,1,Fix1').
   ```

   Since `p` is not considered now in the stratum 1, `Fix1'` can be correctly calculated as

   ```
   Fix1'= {(q(X),true), (q(X),X=a), (q(X),X=b)(r(X),X=c)}.
   ```

2. Forcing the goal `r(X)` with this new fixpoint, by calling

   ```
   force(Delta1,Stratification,Fix1',r(X),C)
   ```

   which computes the constraint `C` as `X=c`.

Then `(p(X), X=c)` is added to the previous interpretation. Since no more pairs can be added,

```
Fix2 = {(p(X),X=c), (q(X),X=a), (q(X),X=b) (r(X),X=c)}
```

is obtained as the final fixpoint for `Delta` we were looking for. □

We finish this section with an example, illustrating how to obtain a dependency graph, in which some negative labeled edges are generated, as explained before, due to an embedded implication.

Example 14. Consider the clause:

$$D \equiv \forall x\big(G \Rightarrow p(x)\big), \quad \text{where } G \equiv \exists y\big(q(x,y) \Rightarrow \big(r(x) \wedge u(y)\big)\big) \wedge \neg t(x).$$

From G the edges $q \to r$ and $q \to u$ are added to the dependency graph. Now G must be connected with the outer context of the clause D, i.e., the predicate symbols q, r, u, t have to be connected with p. This introduces a first edge $q \to p$ and the rest of edges will be negatively labeled: t occurs explicitly negated and r, u occur in a nested implication. So the edges $r \xrightarrow{\neg} p$, $u \xrightarrow{\neg} p$, $t \xrightarrow{\neg} p$ are introduced. Then, collecting all the edges, the dependency graph for D is:

$$\{q \to r, q \to u, q \to p, r \xrightarrow{\neg} p, u \xrightarrow{\neg} p, t \xrightarrow{\neg} p\}.$$

According to Definition 2, a stratification for D is any mapping s from $\{p, q, r, u, t\}$ to natural numbers satisfying $s(q) \leqslant s(r)$, $s(q) \leqslant s(u)$, $s(q) \leqslant s(p)$, $s(r) < s(p)$, $s(u) < s(p)$, $s(t) < s(p)$. For instance, $s(p) = 2$ and $s(q) = s(r) = s(u) = s(t) = 1$. This way, a database with just this clause is stratifiable. Intuitively, this means that for evaluating p, the rest of predicates should be evaluated before; in particular q, which takes part of a nested implication.

But if we add the clause $D' \equiv \forall x \forall y(p(x) \Rightarrow q(x, y))$, the edge $p \to q$ is generated, and we would also have the inequality $s(p) \leqslant s(q)$. Then, the system of inequalities does not have any solution, i.e., the augmented database is not stratifiable. □

The complete specification of the dependency graph used in the implementation (including the additional negatively labeled edges introduced in this section and those introduced by aggregates explained in Section 7.1) and the stratification algorithm can be found in Appendix B.

9. Conclusions and future work

In this work we have presented the formalization and implementation of the constraint logic programming scheme $HH_{\neg}(\mathcal{C})$ as an expressive deductive database language. The main additions of this language w.r.t. the existing ones come from the fact that the intuitionistic logic of hereditary Harrop formulas which it is based on supports implication and universal quantification in goals. This allows to manage hypothetical queries and encapsulation of variables. In addition, $HH_{\neg}(\mathcal{C})$ incorporates the benefits of using constraints. The proposal includes within the same language these features together with stratified negation and a flexible architecture for different constraint systems. In fact, the framework is independent of the concrete constraint system. In particular, we have provided Boolean, Real, and Finite Domains constraint systems. Also, a limited combination of constraints from different constraint systems is allowed. As a result, we obtain a very powerful database language with a well-formalized theoretical framework and a prototype system that implements it.

$HH_{\neg}(\mathcal{C})$ is a mature proposal from the theoretical point of view, and the prototype shows its viability. But the prototype presented in this work must be enhanced to set it as a practical system. The current implementation is very close to the theory and is a valuable tool for understanding and develop such a theory, but as a consequence it has an expected penalty in efficiency.

A first source of inefficiency comes from the forcing of implication, which dynamically changes the database. This involves the local computation of the fixpoint for augmented databases, which yields to start computations from lower strata. Two important improvements oriented to reduce the number of extra computations can be done. First, when solving $D \Rightarrow G$, where $str(D) = i$, D is included into the current database Δ in order to solve G. However, the fixpoint of the augmented database has not to be computed from scratch, but from the known $fix_{i-1}(\Delta)$, that in fact is equal to $fix_{i-1}(\Delta \cup \{D\})$. Second, for computing the answer constraint associated to G, only the semantics of the predicates which G depends on is necessary. These predicates can be identified from the dependency graph, so only the fixpoint for a fragment of the database, corresponding to the clauses defining such predicates, must be computed.

The bottom-up computation by strata we have chosen as computation mechanism offers a number of benefits as we have explained, but it is also a second source of inefficiency. In this line, well-known methods as magic set transformations [6] and tabling [48] could be worth adapting to the current implementation. Therefore, a more efficient top-down-driven bottom-up computation can be achieved, as it is guided by goals. This is also related to widen the set of computable queries and databases that could relax our stratification restrictions. In particular, by using tabling we can avoid the incorporation of negative dependencies to deal with implications in the body of a clause. The idea for adapting tabling to our scheme is not only memorizing answers but also assumed clauses.

In addition, efficiency can be upgraded if some fixpoint components are mapped to relational tables, in this way some operations can be solved with SQL queries by taking advantage of the existing relational technology performance and memory scalability.

There are several interesting extensions that can be incorporated to the system, as strong integrity constraints. This is useful in a number of circumstances, as in Example 6, where it is assumed that each client has, at most, one mortgage quote. This condition could be naturally expressed as an integrity constraint, better than an assumption. A first approach to this issue has already been addressed in [3], where we sketched how to support primary keys, foreign keys and functional dependencies in $HH_{\neg}(\mathcal{C})$. Another possible extension refers to aggregate functions. We have imposed the requirement that every atom over which the aggregate works on must be ground in the interpretation. While this restriction is quite natural and responds to the most common use of aggregates, the possibility of dropping such a restriction and its practical applications can be studied.

Acknowledgements

This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502). Also thanks to Jan Wielemaker for providing SWI-Prolog [53] and Markus Triska [50] for both providing its FD library and adding new features we needed. Finally, we would like to express our thanks to the anonymous referees who really helped us in improving the paper.

Appendix A. Proofs of Section 5

Lemma 2. *Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$ such that $I_1 \sqsubseteq_i I_2$. Then, for any $\Delta \in \mathcal{W}$, and $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, if $I_1, \Delta \models (G, C)$, then $I_2, \Delta \models (G, C)$.*

Proof. The proof is inductive on the structure of G.

(C') This case is trivial by the definition of $\models$.

(A) $I_1, \Delta \models (A, C) \iff (A, C) \in I_1(\Delta)$. In fact $(A, C) \in [I_1(\Delta)]_j$, $1 \leqslant j \leqslant i$. Then, since $I_1 \sqsubseteq_i I_2$ implies that $[I_1(\Delta)]_j \subseteq [I_2(\Delta)]_j \subseteq I_2(\Delta)$, $(A, C) \in I_2(\Delta)$ and therefore $I_2, \Delta \models (A, C)$.

($\neg A$) If $I_1, \Delta \models (\neg A, C)$, then $C \vdash_{\mathcal{C}} \neg C'$ for every C' such that $(A, C') \in I_1(\Delta)$, or there is no such C' and $C \equiv \top$. Since $str(\neg A) \leqslant i$, obviously $str(A) = j$, for some $j < i$. Then $[I_2(\Delta)]_j = [I_1(\Delta)]_j$, because $I_1 \sqsubseteq_i I_2$, and $I_1, \Delta \models (\neg A, C)$ is equivalent to $I_2, \Delta \models (\neg A, C)$.

($G_1 \wedge G_2$) If $I_1, \Delta \models (G_1 \wedge G_2, C)$, then $I_1, \Delta \models (G_1, C)$ and $I_1, \Delta \models (G_2, C)$. In both cases the induction hypothesis can be used, notice that the strata of G_1 and G_2 are less than or equal to i, so $I_2, \Delta \models (G_1, C)$ and $I_2, \Delta \models (G_2, C)$, which implies that $I_2, \Delta \models (G_1 \wedge G_2, C)$.

($G_1 \vee G_2$) $I_1, \Delta \models (G_1 \vee G_2, C) \iff$ there is $k \in \{1, 2\}$ such that $I_1, \Delta \models (G_k, C)$. By induction hypothesis, $I_2, \Delta \models (G_k, C)$, hence $I_2, \Delta \models (G_1 \vee G_2, C)$.

($D \Rightarrow G'$) $I_1, \Delta \models (D \Rightarrow G', C) \iff I_1, \Delta \cup \{D\} \models (G', C)$. Then, by induction hypothesis, $I_2, \Delta \cup \{D\} \models (G', C)$ holds, so $I_2, \Delta \models (D \Rightarrow G', C)$.

($C' \Rightarrow G'$) $I_1, \Delta \models (C' \Rightarrow G', C) \iff I_1, \Delta \models (G', C \wedge C')$. $str(G') = str(C' \Rightarrow G')$, so $str(G') \leqslant i$. Hence, by induction hypothesis, $I_2, \Delta \models (G', C \wedge C')$ holds, which implies that $I_2, \Delta \models (C' \Rightarrow G', C)$.

($\exists x G'$) $I_1, \Delta \models (\exists x G', C) \iff I_1, \Delta \models (G'[y/x], C')$, where y does not occur free in Δ, $\exists x G'$, C, and $C \vdash_{\mathcal{C}} \exists y C'$. By induction hypothesis $I_2, \Delta \models (G'[y/x], C')$, hence $I_2, \Delta \models (\exists x G', C)$.

($\forall x G'$) $I_1, \Delta \models (\forall x G', C) \iff \Delta \models (G'[y/x], C)$, where y does not occur free in Δ, $\forall x G'$, C. By induction hypothesis $I_2, \Delta \models (G'[y/x], C)$, therefore $I_2, \Delta \models (\forall x G', C)$. □

Lemma 3. *Let $i \geqslant 1$ and let $\{I_n\}_{n \geqslant 0}$ be a denumerable family of interpretations over the stratum i, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$. Then, for any Δ, G and C,*

$$\bigsqcup_{n \geqslant 0} I_n, \Delta \models (G, C) \iff \text{there exists } k \geqslant 0 \text{ such that } I_k, \Delta \models (G, C).$$

Proof. In order to simplify the notation we write $\hat{I}$ as $\bigsqcup_{n \geqslant 0} I_n$. The implication from right to left is a consequence of Lemma 2, since $I_k \sqsubseteq_i \hat{I}$ holds for any k. The converse is proved using the result of Lemma 1 ($\hat{I}(\Delta) = \bigcup_{n \geqslant 0} I_n(\Delta)$), by induction on the structure of G.

(C) $\hat{I}, \Delta \models (C, C') \iff I_k, \Delta \models (C, C')$ is true independently of $k \geqslant 0$.

(A) $\hat{I}, \Delta \models (A, C) \iff (A, C) \in \hat{I}(\Delta) = \bigcup_{n \geqslant 0} I_n(\Delta)$. Therefore, there exists $k \geqslant 0$ such that $(A, C) \in I_k(\Delta)$, hence, for that k, $I_k, \Delta \models (A, C)$.

($\neg A$) $\hat{I}, \Delta \models (\neg A, C) \iff$ for every C' such that $\hat{I}, \Delta \models (A, C')$, $C \vdash_{\mathcal{C}} \neg C'$, or there is not such C'. We are assuming that $str(\neg A) \leqslant i$ so $str(A) < i$. $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$ implies that $[I_0(\Delta)]_j = [I_1(\Delta)]_j = \cdots = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j$. So for any $k \geqslant 1$, $I_k, \Delta \models (\neg A, C)$.

($G_1 \wedge G_2$) $\hat{I}, \Delta \models (G_1 \wedge G_2, C) \iff \hat{I}, \Delta \models (G_j, C)$, for each $j \in \{1, 2\}$. In both cases the induction hypothesis can be used, so there exist $k_1, k_2 \geqslant 0$ such that $I_{k_j}, \Delta \models (G_j, C)$ for each $j \in \{1, 2\}$. Let $k = max(k_1, k_2)$. Then $I_k, \Delta \models (G_j, C)$ for each $j \in \{1, 2\}$ in virtue of Lemma 2, and hence $I_k, \Delta \models (G_1 \wedge G_2, C)$.

($G_1 \vee G_2$) $\hat{I}, \Delta \models (G_1 \vee G_2, C) \iff$ there is $j \in \{1, 2\}$ such that $\hat{I}, \Delta \models (G_j, C)$. The induction hypothesis can be used, so there exists $k \geqslant 0$ such that $I_k, \Delta \models (G_j, C)$, and therefore $I_k, \Delta \models (G_1 \vee G_2, C)$.

($D \Rightarrow G'$) $\hat{I}, \Delta \models (D \Rightarrow G', C) \iff \hat{I}, \Delta \cup \{D\} \models (G', C)$. Then there is $k \geqslant 0$ such that $I_k, \Delta \cup \{D\} \models (G', C)$, by induction hypothesis. Therefore there is $k \geqslant 0$ such that $I_k, \Delta \models (D \Rightarrow G', C)$, by definition of the relation $\models$.

($C' \Rightarrow G'$) $\hat{I}, \Delta \models (C' \Rightarrow G', C) \iff \hat{I}, \Delta \models (G', C \wedge C')$. Then there is $k \geqslant 0$ such that $I_k, \Delta \models (G', C \wedge C')$, by induction hypothesis, which means that there is $k \geqslant 0$ such that $I_k, \Delta \models (C' \Rightarrow G', C)$.

($\forall x G'$) $\hat{I}, \Delta \models (\forall x G', C) \iff$ there is a variable y such that y does not occur free in Δ, C, $\forall x G'$, such that $\hat{I}, \Delta \models (G'[y/x], C)$. By induction hypothesis, it happens that there exists $k \geqslant 0$ such that $I_k, \Delta \models (G'[y/x], C)$. Hence $I_k, \Delta \models (\forall x G', C)$ for some $k \geqslant 0$.

($\exists x G'$) $\hat{I}, \Delta \models (\exists x G', C) \iff$ there is a variable y such that y does not occur free in Δ, C, $\exists x G'$, and a constraint C', such that $\hat{I}, \Delta \models (G'[y/x], C')$, and $C \vdash_{\mathcal{C}} \exists y C'$. By induction hypothesis, it happens that there exists $k \geqslant 0$ such that $I_k, \Delta \models (G'[y/x], C')$. Hence, by definition of $\models$, $I_k, \Delta \models (\exists x G', C)$ for some $k \geqslant 0$. □

Proposition 3. *For every $i \geqslant 1$, $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, such that G does not contain negation, if $str(G) \leqslant i$, then:*

$$fix_i, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

Proof. $\Rightarrow$) This implication can be proved by induction on the structural order $(\mathcal{S}_i, <_i)$. Let us take $\langle \Delta, G, C \rangle \in \mathcal{S}_i$ and assume that, for any other $\langle \Delta', G', C' \rangle \in \mathcal{S}_i$, $\langle \Delta', G', C' \rangle <_i \langle \Delta, G, C \rangle$ implies that $\Delta'; C' \vdash_{\mathcal{UC}_\neg} G'$. Then, let us conclude $\Delta; C \vdash_{\mathcal{UC}_\neg} G$ by case analysis on the structure of G.

$C \in \mathcal{SL}_{\mathcal{C}}$ If $\langle \Delta, C', C\rangle \in \mathcal{S}_i$ then $C \vdash_{\mathcal{C}} C'$, therefore $\Delta; C \vdash_{\mathcal{UC}_\neg} C'$ by (C).

$A \in At$ $\langle \Delta, A, C\rangle \in \mathcal{S}_i$ implies that $fix_i, \Delta \Vdash (A, C)$. Let $k = ord(\langle \Delta, A, C\rangle)$, then $T_i^k(fix_{i-1}), \Delta \Vdash (A, C)$, which is equivalent to $(A, C) \in (T_i^k(fix_{i-1}))(\Delta)$. Hence, there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that the variables $\bar{x}$ do not occur free in A, and $T_i^{k-1}(fix_{i-1}), \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. In this case, $\langle \Delta, \exists \bar{x}(A \approx A' \wedge G), C\rangle <_i \langle \Delta, A, C\rangle$, so the induction hypothesis can be applied, obtaining that $\Delta; C \vdash_{\mathcal{UC}_\neg} \exists \bar{x}(A \approx A' \wedge G)$. Using the rule (*Clause*) with the elaborated clause $\forall \bar{x}(G \Rightarrow A')$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} A$.

$G_1 \wedge G_2$ Then $\langle \Delta, G_1 \wedge G_2, C\rangle \in \mathcal{S}_i$ implies that $fix_i, \Delta \Vdash (G_k, C)$ for each $k \in \{1, 2\}$. G_1, G_2 are strict subformulas of $G_1 \wedge G_2$, hence $\langle \Delta, G_k, C\rangle <_i \langle \Delta, G_1 \wedge G_2, C\rangle$, for each $k \in \{1, 2\}$. Then, by the induction hypothesis, $\Delta; C \vdash_{\mathcal{UC}_\neg} G_k$, for each $k \in \{1, 2\}$. So $\Delta; C \vdash_{\mathcal{UC}_\neg} G_1 \wedge G_2$. applying $(\wedge)$ rule.

$G_1 \vee G_2$ Similar to the previous case.

$D \Rightarrow G$ Now $\langle \Delta, D \Rightarrow G, C\rangle \in \mathcal{S}_i$ implies that $fix_i, \Delta \cup \{D\} \Vdash (G, C)$. Clearly, $ord(\langle \Delta, D \Rightarrow G, C\rangle) = ord(\langle \Delta \cup \{D\}, G, C\rangle)$ and G is a strict subformula of $D \Rightarrow G$, so $\langle \Delta \cup \{D\}, G, C\rangle <_i \langle \Delta, D \Rightarrow G, C\rangle$. Therefore, by the induction hypothesis, $\Delta, D; C \vdash_{\mathcal{UC}_\neg} G$. Thanks to the rule $(\Rightarrow)$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} D \Rightarrow G$.

$C' \Rightarrow G$ Then $\langle \Delta, C' \Rightarrow G, C\rangle \in \mathcal{S}_i$ implies that $fix_i, \Delta \Vdash (G, C \wedge C')$. Clearly, $\langle \Delta, G, C \wedge C'\rangle <_i \langle \Delta, C' \Rightarrow G, C\rangle$. Then, by the induction hypothesis, $\Delta; C \wedge C' \vdash_{\mathcal{UC}_\neg} G$. Hence it is easy to show $\Delta; C \vdash_{\mathcal{UC}_\neg} C' \Rightarrow G$, using $(\Rightarrow_C)$ rule.

$\exists xG$ Then $\langle \Delta, \exists xG, C\rangle \in \mathcal{S}_i$ implies that there is C', such that $C \vdash_{\mathcal{C}} \exists yC'$ and $fix_i, \Delta \Vdash (G[y/x], C')$, where y does not occur free in Δ, $\exists xG$ and C. Then $G[y/x]$ is a renaming of a strict subformula of $\exists xG$, and $\langle \Delta, G[y/x], C'\rangle <_i \langle \Delta, \exists xG, C\rangle$. Therefore $\Delta; C' \vdash_{\mathcal{UC}_\neg} G[y/x]$ by the induction hypothesis, so $\Delta; C, C' \vdash_{\mathcal{UC}_\neg} G[y/x]$, trivially. Hence $\Delta; C \vdash_{\mathcal{UC}_\neg} \exists xG$, by using the rule $(\exists)$, because $C \vdash_{\mathcal{C}} \exists yC'$.

$\forall xG'$ Then $\langle \Delta, \forall xG, C\rangle \in \mathcal{S}_i$ implies that $fix_i, w \Vdash (G[y/x], C)$, where the variable y does not occur free in Δ, $\forall xG$, C. Clearly, $ord(\langle \Delta, \forall xG, C\rangle) = ord(\langle \Delta, G[y/x], C\rangle)$ and $G[y/x]$ is a renaming of a strict subformula of $\forall xG$, so $\langle \Delta, G[y/x], C\rangle <_i \langle \Delta, \forall xG, C\rangle$. Therefore, by the induction hypothesis, we obtain $\Delta; C \vdash_{\mathcal{UC}_\neg} G[y/x]$. Applying now $(\forall)$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} \forall xG$.

$\Leftarrow$) It is proved by induction on the height h of the tree proof for $\Delta; C \vdash_{\mathcal{UC}_\neg} G$.

Base case: $h = 1$. The only possibility is that $G \equiv C' \in \mathcal{SL}_{\mathcal{C}}$. Obviously $fix_i, \Delta \Vdash (C', C)$, because we are assuming $\Delta; C \vdash_{\mathcal{UC}_\neg} C'$, so $C \vdash_{\mathcal{C}} C'$.

Inductive case: We suppose that $\Delta; C \vdash G$ has a proof of height h. Let us prove that $fix_i, \Delta \Vdash (G, C)$, by case analysis on the rule employed in the bottom of such proof.

(*Clause*) There must exist a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that $\bar{x}$ do not occur free in A, and that $\Delta; C \vdash \exists \bar{x}(A \approx A' \wedge G)$ has a proof of height $h - 1$. By induction hypothesis, $fix_i, \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. Using the definition of the operator T_i, the latter implies $(A, C) \in (T_i(fix_i))(\Delta) = fix_i(\Delta)$, then $fix_i, \Delta \Vdash (A, C)$.

$(\wedge)$ There must exist goals G_1, G_2 such that $G \equiv G_1 \wedge G_2$ and the sequent $\Delta; C \vdash G_k$ has a proof of height less than h for each $k \in \{1, 2\}$. By induction hypothesis, $fix_i, \Delta \Vdash (G_1, C)$, and $fix_i, \Delta \Vdash (G_2, C)$. As a consequence, $fix_i, \Delta \Vdash (G_1 \wedge G_2, C)$.

$(\vee)$ Similar to the previous case.

$(\Rightarrow)$ Then $G \equiv D \Rightarrow G'$ and the sequent $\Delta, D; C \vdash G'$ has a proof of height $h - 1$. By induction hypothesis, $fix_i, \Delta \cup \{D\} \Vdash (G', C)$. Therefore $fix_i, \Delta \Vdash (D \Rightarrow G', C)$.

$(\Rightarrow_C)$ Now $G \equiv C' \Rightarrow G'$ and the sequent $\Delta; C, C' \vdash G'$ has a proof of height $h-1$. By the properties of $\vdash_{\mathcal{UC}_\neg}$, also $\Delta; C \wedge C' \vdash G'$ has a proof of height $h - 1$. Applying now the induction hypothesis, $fix_i, \Delta \Vdash (G', C \wedge C')$. Therefore $fix_i, \Delta \Vdash (C' \Rightarrow G', C)$.

$(\exists)$ Then $G \equiv \exists xG'$, and there must exist a constraint C' and a variable y not occurring free in Δ, C, $\exists xG'$, such that $\Delta; C \wedge C' \vdash G'[y/x]$ has a proof of height $h - 1$ and $C \vdash_{\mathcal{C}} \exists yC'$. Then $fix_i, \Delta \Vdash (G'[y/x], C \wedge C')$, by induction hypothesis, and therefore $fix_i, \Delta \Vdash (\exists xG', C)$, because $C \vdash_{\mathcal{C}} \exists y(C \wedge C')$.

$(\forall)$ G must be of the form $\forall xG'$, and there must exist a variable y not occurring free in Δ, C, $\forall xG'$ such that $\Delta; C \vdash G'[y/x]$, has a proof of height $h - 1$. By induction hypothesis, $fix_i, \Delta \Vdash (G'[y/x], C)$, and, as a consequence, $fix_i, \Delta \Vdash (\forall xG', C)$. □

Appendix B. Dependency graph and stratification

The algorithm for calculating the dependency graph is expressed by means of the mutually recursive functions *dpClause* and *dpGoal* defined in Fig. 7, depending on the structure of the formula. These functions return a pair $\langle E, N\rangle$, where E is a set of edges of the form $p \to q$ or $p \xrightarrow{\neg} q$, and N is an auxiliary set of *link*-nodes which stores the predicate symbols occurring in the formula. Moreover, each of these predicate symbols has a negative annotation in three cases: if they occur in a negated atom, in a nested implication or in an aggregate function (notice that the link-nodes, $\neg preds(C)$, in the fourth case in the definition of *dpGoal* correspond to the aggregate functions occurring in C). This annotation will be then propagated to the edges coming out of those nodes.

By using the function *dpClause* and *dpGoal*, it is straightforward to calculate the dependency graph of a set of formulas Φ (and in particular, for a database) as the union of the edges obtained for each element of the set:

- $dpClause(A) = \langle \emptyset, \{p_A\}\rangle$
- $dpClause(D_1 \wedge D_2) = \langle E_1 \cup E_2, N_1 \cup N_2\rangle$
  if $dpClause(D_1) = \langle E_1, N_1\rangle$ and $dpClause(D_2) = \langle E_2, N_2\rangle$
- $dpClause(\forall x\ D) = dpClause(D)$
- $dpClause(G \Rightarrow A) = \langle E_G \cup \bigcup_{n\in N_G}\{n \to p_A\} \cup \bigcup_{\neg n\in N_G}\{n \xrightarrow{\neg} p_A\}, \{p_A\}\rangle$
  if $dpGoal(G) = \langle E_G, N_G\rangle$

- $dpGoal(A) = \langle \emptyset, \{p_A\}\rangle$
- $dpGoal(\neg A) = \langle \emptyset, \{\neg p_A\}\rangle$
- $dpGoal(C) = \langle \emptyset, \neg preds(C)\rangle$
- $dpGoal(C \Rightarrow G) = \langle E \cup \bigcup_{n\in preds(C),\ m\in preds(G)}\{n \xrightarrow{\neg} m\}, N \cup \neg preds(C)\rangle$
  if $dpGoal(G) = \langle E, N\rangle$
- $dpGoal(G_1 \wedge G_2) = dpGoal(G_1 \vee G_2) = \langle E_1 \cup E_2, N_1 \cup N_2\rangle$
  if $dpGoal(G_1) = \langle E_1, N_1\rangle$ and $dpGoal(G_2) = \langle E_2, N_2\rangle$
- $dpGoal(\forall x\, G) = dpGoal(\exists x\, G) = dpGoal(G)$
- $dpGoal(D \Rightarrow G) = \langle E_D \cup E_G \cup \bigcup_{m\in preds(G)}(\bigcup_{n\in N_D}\{n \to m\} \cup \bigcup_{\neg n\in N_D}\{n \xrightarrow{\neg} m\}), N_D \cup \neg preds(G)\rangle$
  if $dpClause(D) = \langle E_D, N_D\rangle$ and $dpGoal(G) = \langle E_G, N_G\rangle$

Notation:

- p_A: predicate symbol of the atom A
- $preds(F) = \{p \mid p$ is a definite predicate symbol occurring in $F\}$
- $\neg S = \{\neg p \mid p \in S\}$

Fig. 7. Dependency graph for clauses and goals.

$$DG_\Phi = \bigcup_{D\in\Phi}\big\{E_D \mid dpClause(D) = \langle E_D, N\rangle\big\} \cup \bigcup_{G\in\Phi}\big\{E_G \mid dpGoal(G) = \langle E_G, N\rangle\big\}.$$

Once we have the dependency graph, the particular algorithm for finding a stratification for Δ (or for checking that it is not stratifiable) associates to each predicate symbol p an integer variable $X_p \in [1 \ldots N]$, where N is the number of predicate symbols of Δ, and generates a system of inequalities: each dependency $p \to q$ produces $X_p \leqslant X_q$ and $p \xrightarrow{\neg} q$ produces $X_p < X_q$. Then, solving this system (if possible) provides the stratum of each p in X_p. The stratification algorithm ends with a concrete stratification if there exists one or stops with an error message (in a polynomial time with respect to the number of predicate symbols in the database).

Appendix C. Implementation of constraint solving

This appendix includes implementation details of the constraint solvers, focusing on the finite domain and aggregates.

C.1. Implementing `solve`

The generic interface to the constraint solvers is implemented as follows:

```
solve(I,C,SC) :-
  simplify_ground_ctr(C,SGC),
  partition_ctr(I,SGC,DCs),
  solve_ctr_list(I,DCs,SDCs),
  ctr_list_to_ctr(SDCs,CC),
  simplify_ctr(CC,SC).
```

This code first calls `simplify_ground_ctr`, which simplifies trivial ground primitive constraints (as, e.g., $=, >, \ldots$). Next, the call to predicate `partition_ctr` partitions the input constraint into a list whose components belong to different constraint domains. This partition is always possible when the constraint ranges over a single domain. When it combines different domains, the current implementation is able to achieve the partition in some cases. If it is unable to do it, then an exception is raised.

The call to `solve_ctr_list` posts each component to its corresponding solver as a call to the predicate `solveFD` (described later). After, the solved constraint, which is represented as a list, is transformed back into a conjunctive constraint via `ctr_list_to_ctr`. Finally, this constraint is simplified by logical axioms as De Morgan's laws. In addition to this generic interface, the particular interface

```
solve(+Domain,+Interpretation,+Constraint,-SolvedConstraint).
```

is also provided, which is useful when the domain `Dom` is already known and can be directly posted to its corresponding solver.

```
(01) solveFD(Dom,I,C,SC) :-
(02)   copy_term(C,FC),                  % Input variables keep untouched
(03)   get_vars(C,Vars),                 % Input variables are held to be
(04)   get_vars(FC,FVars),               %  mapped to the solved new vars
(05)   swap_qvars_by_fvars(FC,QFC),      % Replace quantified vars by fresh ones
(06)   constrain_domains(QFC,Dom),       % Constrain variables to the current domain
(07)   domain_to_int(QFC,Dom,IC),        % Domain mapping from enumerated to integer
(08)   bagof((FVars,Cs,Sat),             % List of (Fresh vars,Constraints,Satisfiable)
(09)     (solveFD_ctr(IC,Dom,I,true),    % Solving
(10)      satisfiable(IC,Sat),           % Check satisfiability
(11)      project_ctrs(FVars,Vars,Cs)    % Project constraints wrt. input vars
(12)     ), LFVarsCsS), !                % List of Fresh vars,Constraints,Satisfiable
(13)   filter_ctr_list(LFVarsCsS,LICs), % Pick solved constraints
(14)   simplify_disj_list(LICs,SLICs),   % Simplify the disjunctive list
(15)   disj_list_to_ctr(SLICs,ISC),      % Convert list to constraint
(16)   get_vars(ISC,FVSC),               % If the output constraint contains
(17)   (fresh_vars(Vars,FVSC) ->         %  fresh variables
(18)    SC = C                           % Then discard the solved constraint
(19)    ;                                %  and return the input constraint
(20)    int_to_domain(ISC,Dom,SC)).      % Else, map domain from integer to enumerated
(21) solveFD(_Dom,_I,_C,false).          % Return false upon unsatisfiability
```

Fig. 8. The Predicate `solveFD` for solving finite domain constraints.

C.2. Implementing `solveFD`

For the solvers of the constraint systems Finite Domains and Boolean, the following predicates are available:

- `solveFD(+Domain,+Interpretation,+Constraint,-SolvedConstraint)`
  It solves the input `Constraint` over `Domain` using `Interpretation` and returns its solved form `Solved Constraint`, if it is satisfiable; otherwise, returns `false`.
- `constr_conjFD(+Domain,+Interpretation,-C1,+C2,+C)`
  It computes (using `Interpretation`) the component `C1` of the conjunction `C1,C2` such that `C1,C2` $\vdash_{\mathcal{C}}$ `C`, where $\mathcal{C}$ is the constraint system corresponding to `Domain`.

Since we consider classical logic for these particular constraint systems, the following implementation for the second predicate is sound:

```
constr_conjFD(Dom,I,C1,C2,C) :-
  solveFD(Dom,I,(not(C2);C),C1), !,
  C1\==false.
```

Predicate `solveFD` includes calls to the solving of constraints, which are computed with the predicate:

```
solveFD_ctr(+Constraint,+DomainName,+Interpretation,-Satisfiable),
```

which receives a constraint, a domain name and an interpretation, returning whether it is satisfiable (`true`) or not (`false`).

The code excerpt of Fig. 8 implements the required behavior for `solveFD`. Line `(05)` is intended to replace quantified variables by fresh ones in order to avoid a name clash. Line `(07)` maps domain data values with integers, whereas line `(21)` replaces back the (integer) computed data values by the corresponding, mapped data values. The core of constraint solving lays between lines `(09)-(11)`, where, first, the input constraint is tried to be solved (see next paragraph describing the predicate `solveFD_ctr`). Second, it is checked for satisfiability, that is, try to find a single, concrete solution via labeling. And, third, the underlying constraint store is projected with respect to the relevant variables (i.e., those occurring in the input constraint plus the possible new ones computed by the underlying solver). Lines `(13)-(15)` are simply intended for data structure formatting.

C.2.1. Aggregates

During evaluating expressions, the predicate `compute_aggr` is responsible of computing the outcome of each aggregate function, as illustrated by one of its clauses below:

```
compute_aggr(sum(At,Var),I,Res) :-
   !,
   get_values(I,At,Var,S),
   compute_sum(S,0,Res).
```

The predicate `get_values` calls the predicate `lookUpAll` to get concrete values from ground equalities of the form `Var = value` from an interpretation `I`, with respect to an atom `At` and a variable `Var`. Then, these values are stored in a list `S`. Finally, `S` is used to compute the final result `Res` w.r.t. the particular function.

The implementation of the function `count` is slightly different from the others because the call to this function has not a variable as an argument. In this case, we also call the predicate `lookUpAll`, but then, the solver computes the number of occurrences of the atom to be counted from the interpretation.

C.3. Solving primitive constraints

Whereas some constraints can be posted to the underlying solver, others cannot, as negation which is, as shown below, explicitly handled because it can apply to constraints that are not supported by the underlying Prolog solver:

```
solveFD_ctr(not(C),Dom,I,B) :-
  !,
  complement(C,NotC),
  solveFD_ctr(NotC,Dom,I,B).
```

Here, the predicate `complement` computes the complemented constraint (e.g., `X#=<Y` is the complemented constraint of `X#>Y`).

An example of unsupported constraint is disjunction, which is computed by collecting all answers (cf. line `(08)` in Fig. 8). Solving this constraint is as follows:

```
solveFD_ctr((C1;_C2),Dom,I,true) :-
  solveFD_ctr(C1,Dom,I,true).
solveFD_ctr((_C1;C2),Dom,I,true) :-
  !,
  solveFD_ctr(C2,Dom,I,true).
```

Finally, we describe quantifiers. The existential quantifier is implemented as follows, where in the last but one line `satisfiable(FC,Domain,true)` tries to find a concrete value satisfying `FC`:

```
solveFD_ctr(ex(X,C),Dom,I,B) :-
  !,
  % Replace X by a fresh variable _FX in C:
  swap(X,_FX,C,FC),
  constrain_domains(FC,Dom),
  (solveFD_ctr(FC,Dom,I,true),
   % Checking satisfiability:
   satisfiable(FC,Dom,true),
   B=true
   ;
   B=false
  ).
```

For the universal quantifier, a constraint `fa(X,C)` is replaced by the conjunction `C[X/v1], ..., C[X/vn]`, where `v`i ($1 \leqslant i \leqslant n$) are the values in the domain of `X`. Note that cuts at the beginning in the body of each clause occur because we add a default case corresponding to an illegal constraint, which involves the raising of an exception.

The constraint solver for Reals follows a similar but simpler route for its implementation since universal quantifiers are not supported, and there are no domain data values to map. Both `solveR` and `constr_conjR` are provided, analogously to `solveFD` and `constr_conjFD` respectively.

Appendix D. Implementation of the forcing relation

The forcing relation $\Vdash$ of Definition 6 is implemented by means of the predicate

```
force(+Delta,+Stratification,+I,+G,-C)
```

whose meaning is: given `I` $= T_i^n(fix_{i-1})$(`Delta`), for some $n \geqslant 0$ and a fixed stratum $i > 0$, `force` is successful if $T_i^n(fix_{i-1})$, `Delta` $\Vdash$ (`G`, `C`). An important point to understand the implementation is to keep in mind the deterministic nature of this predicate. The definition of $\Vdash$ establishes conditions on a constraint C in order to satisfy `I`, `Delta` $\Vdash$ (`G`, `C`), but the predicate `force` must build a concrete constraint `C`. In addition, each possible answer constraint for a goal must be captured in a single answer constraint (possibly) using disjunctions.

```
(1) force(_Delta,_Stratification,I,constr(Dom,C),C1):- !,
      solve(Dom,I,C,C1).

(2) force(Delta,Stratification,I,(G1,G2),C):- !,
      force(Delta,Stratification,I,G1,C1),
      force(Delta,Stratification,I,G2,C2),
      solve(I,(C1,C2),C).

(3) force(Delta,Stratification,I,(G1;G2),C):- !,
      ( force(Delta,Stratification,I,G1,C1), !,
        ( force(Delta,Stratification,I,G2,C2), !, solve(I,(C1;C2),C)
        ; solve(I,C1,C) )
      ; force(Delta,Stratification,I,G2,C2), solve(I,C2,C) ).

(4) force(Delta,Stratification,I,(constr(D,C2) => G),C1):- !,
      ( solve(D,I,C2,_), !, force(Delta,Stratification,I,G,C),
        constr_conj(D,I,C1,C2,C)
      ; C1=true ).

(5) force(Delta,Stratification,I,(D => G),C) :- !,
      elab(D,De),
      localClauses(De,Ls), addLocalClauses(Ls,Delta,Delta1),
      getMaxStrat(G,Stratification,StG),
      fixPointStrat(Delta1,Stratification,StG,Fix),
      force(Delta1,Stratification,Fix,G,C).

(6) force(Delta,Stratification,I,ex(X,G),C):- !, replace(X,X1,G,G1),
      force(Delta,Stratification,I,G1,C1), solve(I,ex(X1,C1),C).

(7) force(Delta,Stratification,I,fa(X,G),C):- !, replace(X,X1,G,G1),
      force(Delta,Stratification,I,G1,C1), solve(I,fa(X1,C1),C).

(8) force(_Delta,_Stratification,I,not(At),C):- !, lookUpAll(At,I,Ls),
      ( Ls==[], !, C=true ; buildNegConj(Ls,NLs), solve(I,NLs,C) ).

(9) force(_Delta,_Stratification,I,At,C):-
      lookUpAll(At,I,Cs), buildDisj(Cs,C1), solve(I,C1,C).
```

Fig. 9. Forcing relation.

Fig. 9 shows the implementation of `force`. There is a clause of `force` for each goal structure. We explain them shortly.

Clause `(1)` stands for the forcing of a constraint `C` over a domain `Dom`, which is processed by calling the constraint solver. Clause `(2)` stands for a conjunction `G1,G2`; it forces both goals, and then solves the conjunction of the resulting answer constraints. For a disjunction `G1;G2` (clause `(3)`) there are four possible (and mutually exclusive) situations: both goals can be forced, only `G1`, only `G2`, or neither of two; the answer constraint is obtained by solving the corresponding constraints or failing in the last case.

Clause `(4)` corresponds to an implication with a constraint as antecedent. In this case, we first try to solve the antecedent `C2` in order to check its satisfiability. If it is not satisfiable (second part of the disjunction) then the implication trivially holds and the answer constraint `C1` is `true`. If it is satisfiable, then the consequent `G` must be forced, obtaining an answer constraint `C`. This answer constraint is obtained using the predicate `constr_conj` in order to find `C1` such that $\texttt{C1},\texttt{C2} \vdash_{\mathcal{C}} \texttt{C}$ as stated in the theory (see Definition 6).

Clause `(5)` corresponds to the case of an implication `D => G` which introduces additional difficulties as explained in Section 8.1, as it involves a computation of a new fixpoint for the extended database. The predicate `elab` provides the rules corresponding to the elaboration of the clause `D` as explained in Section 3.1, then `localClauses` transforms them into the used representation `hhcnClause(Vars,Head,Body)`. Calling to `addLocalClauses` the extended database $\texttt{Delta1} = \texttt{Delta} \cup \{\texttt{D}\}$ is obtained. The execution of

```
fixPointStrat(Delta1,Stratification,StG,Fix)
```

finds $\texttt{Fix} = \mathit{fix}_{\texttt{StG}}(\texttt{Delta1})$.

Once `Fix` is computed, it is used to force `G` with the augmented set `Delta1`. This corresponds to prove $\texttt{force}(\texttt{Delta1},\texttt{Stratification},\texttt{Fix},\texttt{G},\texttt{C})$, which implies $T_i^n(I'), \texttt{Delta} \cup \{\texttt{D}\} \models (\texttt{G},\texttt{C})$, which is what we wanted to prove.

For the existential (clause `(6)`), according to the definition of $\models$, to find `C` such that

$$\texttt{I},\texttt{Delta} \models \big(\texttt{ex(X,G)},\texttt{C}\big)$$

(analogously for `fa(X,G)`, clause `(7)`), we obtain `G1` as the result of replacing `X` by a new variable `X1` in `G`; then we prove $\texttt{I},\texttt{Delta} \models (\texttt{G1},\texttt{C1})$, and finally `C` is obtained by solving $\texttt{ex}(\texttt{X1},\texttt{C1})$ ($\texttt{fa}(\texttt{X1},\texttt{C1})$, respectively).

For a negated atom `not(At)` (clause `(8)`), thanks to the stratification, we can ensure that every possible atom `At` deducible from the database is already present in the current interpretation `I`. Then, by means of `lookUpAll(At,I,Ls)`, we find the list `Ls=[C1,...,Cn]` such that `(At,Ci) ∈ I`. The variable `NLs` is used to build the constraint `not(C1),...,not(Cn)` (or `true` if `Ls=[]`), that we must solve to obtain the constraint `C` we are looking for.

Clause `(9)` (default case) is the forcing of an atom `At`. As before, we search for all the pairs `(At,C1),...,(At,Cn) ∈ I` and then we build the disjunction `C1=C1;...;Cn` and solve it with `solve`.

References

[1] K. Apt, R. Bol, Logic programming and negation: A survey, J. Log. Program. 19 (1994) 9–71.
[2] G. Aranda, S. Nieva, F. Sáenz-Pérez, J. Sánchez, Implementing a fixpoint semantics for a constraint deductive database based on hereditary Harrop formulas, in: Proceedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing, PPDP'09, ACM Press, 2009, pp. 117–128.
[3] G. Aranda, S. Nieva, F. Sáenz-Pérez, J. Sánchez, Incorporating integrity constraints to a deductive database system, in: XI Jornadas sobre Programación y Lenguajes, PROLE'11, 2011, pp. 141–152.
[4] V. Bárány, B. ten Cate, M. Otto, Queries with guarded negation, in: Proceedings of the VLDB Endowment, vol. 5, 2012, pp. 1328–1339.
[5] R. Barbuti, M. Martelli, A tool to check the non-floundering logic programs and goals, in: Proceedings of the Programming Language Implementation and Logic Programming 1st International Workshop, PLILP'88, in: LNCS, vol. 348, Springer, 1988, pp. 58–67.
[6] C. Beeri, R. Ramakrishnan, On the power of magic, J. Log. Program. 10 (1991) 255–299.
[7] M. Benedikt, L. Libkin, Safe constraint queries, in: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'98, ACM Press, 1998, pp. 99–108.
[8] N. Bidoit, Negation in rule-based database languages: A survey, Theoret. Comput. Sci. 78 (1) (1991) 3–83.
[9] A. Bonner, Hypothetical datalog: Complexity and expressibility, Theoret. Comput. Sci. 76 (1) (1990) 3–51.
[10] A. Bonner, A logical semantics for hypothetical rulebases with deletion, J. Log. Program. 32 (2) (1997) 119–170.
[11] J.H. Byon, P.Z. Revesz, DISCO: A constraint database system with sets, in: Proceedings of the Workshop on Constraint Databases and Applications, in: LNCS, vol. 1034, Springer-Verlag, 1995, pp. 68–83.
[12] M. Cai, D. Keshwani, P.Z. Revesz, Parametric rectangles: A model for querying and animation of spatiotemporal databases, in: Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology, EDBT'00, in: LNCS, vol. 1777, Springer-Verlag, 2000, pp. 430–444.
[13] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer-Verlag, 1990.
[14] D. Chan, An extension of constructive negation and its application in coroutining, in: Proceedings of the North American Conference on Logic Programming, NACLP'89, MIT Press, 1989, pp. 477–493.
[15] H. Christiansen, T. Andreasen, A practical approach to hypothetical database queries, in: Transactions and Change in Logic Databases, in: LNCS, vol. 1472, Springer, 1998, pp. 340–355.
[16] W. Drabent, What is failure? An approach to constructive negation, Acta Inform. 32 (1995) 27–29.
[17] D.M. Gabbay, N-prolog: An extension of prolog with hypothetical implication II – Logical foundations, and negation as failure, J. Log. Algebr. Program. 2 (1985) 251–283.
[18] M. García-Díaz, S. Nieva, Solving constraints for an instance of an extended CLP language over a domain based on real numbers and Herbrand terms, J. Funct. Log. Program. 2 (2003).
[19] M. García-Díaz, S. Nieva, Providing declarative semantics for HH extended constraint logic programs, in: Proceedings of the 6th ACM SIGPLAN International Conference of Principles and Practice of Declarative Programing, PPDP'04, ACM Press, 2004, pp. 55–66.
[20] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.
[21] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the International Conference on Logic Programming/Symposium on Logic Programming, MIT Press, 1988, pp. 1070–1080.
[22] M.F. Goodchild, Twenty years of progress: Giscience in 2010, J. Spatial Inform. Sci. 1 (2010) 3–20.
[23] R. Gross, Implementation of constraint database systems using a compile-time rewrite approach, PhD thesis, ETH, Zurich, 1996.
[24] J. Harland, Success and failure for hereditary Harrop formulae, J. Log. Program. 17 (1993) 1–29.
[25] J. Jaffar, J.L. Lassez, Constraint logic programming, in: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, POPL'87, ACM Press, 1987, pp. 111–119.
[26] J. Jaffar, M.J. Maher, Constraint logic programming: A survey, J. Log. Program. 19/20 (1994) 503–581.
[27] P. Kanellakis, G. Kuper, P. Revesz, Constraint query languages, J. Comput. System Sci. 51 (1995) 26–52.
[28] P. Kanjamala, P.Z. Revesz, Y. Wang, MLPQ/GIS: A GIS using linear constraint databases, in: Proceedings of the Ninth International Conference on Management of Data, McGraw–Hill, 1998, pp. 389–393.
[29] K. Kunen, Negation in logic programming, J. Log. Program. 4 (1987) 289–308.
[30] G. Kuper, L. Libkin, J. Paredaens, Constraint Databases, Springer, 2000.
[31] J. Leach, S. Nieva, M. Rodríguez-Artalejo, Constraint logic programming with hereditary Harrop formulas, Theory Pract. Log. Program. 1 (2001) 409–445.
[32] V. Lifschitz, Introduction to answer set programming, in: Introductory course at the 16th European Summer School in Logic, Language and Information, Unpublished draft, available at: www.cs.utexas.edu/users/vl/mypapers/esslli.ps, 2004.
[33] J. Lipton, S. Nieva, Higher-order logic programming languages with constraints: A semantics, in: Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications, in: LNCS, vol. 4583, Springer-Verlag, 2007, pp. 272–289.
[34] J.W. Lloyd, Foundations of Logic Programming, Springer, 1987.
[35] D. Miller, G. Nadathur, F. Pfenning, A. Scedrov, Uniform proofs as a foundation for logic programming, Ann. Pure Appl. Logic 51 (1991) 125–157.
[36] D. Miller, G. Nadathur, A. Scedrov, Hereditary Harrop formulas and uniform proof systems, in: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, LICS'87, IEEE Computer Society, 1987, pp. 98–105.
[37] A. Momigliano, Minimal negation and hereditary Harrop formulae, in: Proceedings of the Logical Foundations of Computer Science (LFCS), in: LNCS, vol. 620, Springer, 1992, pp. 326–335.
[38] S. Nieva, F. Sáenz-Pérez, J. Sánchez, Formalizing a constraint deductive database language based on hereditary Harrop formulas with negation, in: Proceedings of the International Symposium on Functional and Logic Programming, FLOPS'08, in: LNCS, vol. 4989, Springer-Verlag, 2008, pp. 289–304.
[39] K. Ramamohanarao, J. Harland, An introduction to deductive database languages and systems, VLDB J. 3 (1994) 107–122.
[40] P.Z. Revesz, Safe datalog queries with linear constraints, in: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming, CP98, in: LNCS, vol. 1520, Springer, 1998, pp. 355–369.
[41] P.Z. Revesz, Safe query languages for constraint databases, ACM Trans. Database Syst. 23 (1998) 58–99.

[42] P.Z. Revesz, Introduction to Constraint Databases, Springer, 2002.
[43] P. Revesz, MLPQ/Presto Users' Manual, Department of Computer Science and Engineering, University of Nebraska–Lincoln, 2004.
[44] P. Rigaux, M. Scholl, A. Voisard, Spatial Database: With Application to GIS, Morgan Kaufmann Ser. Data Manage. Syst., Morgan Kaufmann, 2002.
[45] V.A. Saraswat, The category of constraint systems is cartesian-closed, in: Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, LICS'92, IEEE Computer Society, 1992, pp. 341–345.
[46] P. Stuckey, Negation and constraint logic programming, Inform. and Comput. 118 (1995) 12–33.
[47] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer, Aggregate functions in DLV, in: Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 2nd International ASP'03 Workshop, 2003, CEUR-WS.org.
[48] H. Tamaki, T. Sato, Old resolution with tabulation, in: Proceedings of the 3rd International Conference on Logic Programming, ICLP'86, in: LNCS, vol. 255, 1986, pp. 84–98.
[49] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific J. Math. 5 (1955) 285–309.
[50] M. Triska, Generalising constraint solving over finite domains, in: Proceedings of the 24th International Conference on Logic Programming, ICLP'08, in: LNCS, vol. 5366, Springer-Verlag, 2008, pp. 820–821.
[51] J. Ullman, Database and Knowledge-Base Systems, vols. I (Classical Database Systems) and II (The New Technologies), Computer Science Press, 1995.
[52] A. Van Gelder, K.A. Ross, J.S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (1991) 619–649.
[53] J. Wielemaker, An overview of the SWI-prolog programming environment, in: Proceedings of the 13th International Workshop on Logic Programming Environments, Katholieke Universiteit Leuven, Department of Computer Science, 2003, pp. 1–16.
[54] C. Zaniolo, Key constraints and monotonic aggregates in deductive databases, in: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II, Springer-Verlag, 2002, pp. 109–134.
[55] C. Zaniolo, N. Arni, K. Ong, Negation and aggregates in recursive rules: The LDL++ approach, in: Proceedings of the International Conference on Deductive and Object-Oriented Databases, DOOD, Springer, 1993, pp. 204–221.
[56] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, R. Zicari, Advanced Database Systems, Morgan Kaufmann Publishers, 1997.

Capítulo 6

Publicaciones asociadas al tercer capítulo

[B.1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Formalizing a Broader Recursion Coverage in SQL.
En *Symposium on Practical Aspects of Declarative Languages (PADL'13)*, volumen 7752 de *LNCS*, páginas 93 – 108, 2013.
→ **Página** 176

[B.2] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
Incorporating Hypothetical Views and Extended Recursion into SQL Database Systems.
En Ken Mcmillan, Aart Middeldorp, Geoff Sutcliffe, y Andrei Voronkov, editores, *LPAR-19*, volumen 26 de *EPiC Series*, páginas 9–22. EasyChair, 2014.
→ **Página** 192

[B.3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández.
R-SQL: An SQL Database System with Extended Recursion.
En *Electronic Communications of the EASST*, volumen 64: Programming and Computer Languages, páginas 1–18, 2013.
→ **Página** 206

Formalizing a Broader Recursion Coverage in SQL

Gabriel Aranda[1], Susana Nieva[1], Fernando Sáenz-Pérez[2], and Jaime Sánchez-Hernández[1] *

[1] Dept. Sistemas Informáticos y Computación, UCM, Spain
[2] Dept. Ingeniería del Software e Inteligencia Artificial, UCM, Spain
garanda@fdi.ucm.es, {nieva,fernan,jaime}@sip.ucm.es

Abstract. SQL is the *de facto* standard language for relational databases and has evolved by introducing new resources and expressive capabilities, such as recursive definitions in queries and views. Recursion was included in the SQL-99 standard, but this approach is limited as only linear recursion is allowed, mutual recursion is not supported, and negation cannot be combined with recursion. In this work, we propose a new approach, called R-SQL, aimed to overcome these limitations and others, allowing in particular cycles in recursive definitions of graphs and mutually recursive relation definitions. In order to combine recursion and negation, we import ideas from the deductive database field, such as stratified negation, based on the definition of a dependency graph between relations involved in the database. We develop a formal framework using a stratified fixpoint semantics and introduce a proof-of-concept implementation.

Keywords: Databases, SQL, Recursion, Fixpoint Semantics

1 Introduction

Codd's famous paper on relational model [2] sowed the seeds for current relational database management systems (RDBMS's), such as DB2, Oracle, MySQL, SQL Server and others. Formal query languages were proposed for the relational model: Relational algebra (RA) and relational calculus, which are syntactically different but semantically equivalent w.r.t. safe formulas [16]. Such RDBMS's rather rely on the SQL query language (current standard SQL:2008 [7]) that departs from the relational model and goes beyond. Its acknowledged success builds upon an elegant and yet simple formulation of a data model with relations which can be queried with a language including some basic RA-operators, which are all about relations. Original operators became a limitation for practical applications of the model, and others emerged to fill some gaps, including, for instance, aggregate operators for, e.g., computing running sums and averages.

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

Other additions include representing absent or unknown information, which delivered the introduction of null values and outer join operators ranging over such values. Also, duplicates were introduced to account for bags (multisets) instead of sets. Finally, we mention the inclusion of recursion (Starburst [10] was the first non-commercial RDBMS to implement this whereas IBM DB2 was the first commercial one), a powerful feature to cope with queries that must be otherwise solved by the intermixing with a host language. However, as pointed out by many (see, e.g., [9],[13]), the relational model has several limitations. Thus, such current RDBMS's include that extended "relational" model, which is far from the original one and it is even heavily criticized [3] because of nulls and duplicates.

In this work, we focus on the inclusion of recursion in SQL as current RDBMS's lack both a formal support and suffer a narrow coverage of recursion. Regarding formalization, an extension of the RA is presented in [1], with a looping construct and assignment in order to deal with the integration of recursion and negation. [5] is the source of the original SQL-99 proposal for recursion, which is based on the research in the areas of logic programming and deductive databases [16], as explained in [4]. Another example of an approach built on an extension of RA with a fixpoint construct is in [6]. However, as far as we know, these formalizations do not lead to concrete implementations, while our proposal provides an operational mechanism allowing a straightforward implementation.

Regarding recursion coverage, there are several main drawbacks in current implementations of recursion: Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Some other features are not supported: Mutual recursion, and query solving involving an EXCEPT clause. In general, termination is manually controlled by limiting the number of iterations instead of detecting that there are no further opportunities to develop new tuples.

Here, we propose R-SQL, a subset of the SQL standard to cope with recursive definitions which are not restricted as current RDBMS's do, and also allowing neater formulations by allowing concise relation definitions (much following the assignment RA-operator) and avoiding extensive writings (cf. Section 2). For this language, first we develop a novel formalization based on stratified interpretations and a fixpoint operator to support theoretical results (cf. Section 3). And, second, we propose a proof-of-concept implementation which takes a set of database relation (in general, recursive) definitions and computes their meanings (cf. Section 4). This implementation uses the underlying host SQL system and Python to compute the outcome, and can be easily adapted to be integrated as a part of any state-of-the-art RDBMS. Section 5 concludes and presents some further work.

2 Introducing R-SQL

In this section, we present the language R-SQL by using a minimal syntax that allows to capture the core expressiveness of standard SQL. Namely, we consider

basic SQL constructs to cover relational algebra. Nevertheless, this language is conceived to be able to be extended in order to incorporate other usual features. R-SQL is focused on the incorporation of recursive relation definitions. The idea is simple and effective: A relation is defined with an assignment operation as a named query (view) that can contain a self reference, i.e., a relation `R` can be defined as `R sch :=` SELECT ... FROM ... `R` ..., where `sch` is the relation schema. Next, we introduce the formal grammar of this language, then we show by means of examples the benefits of R-SQL w.r.t. current RDBMS systems.

2.1 Syntax of R-SQL

The formal syntax of R-SQL is defined by the grammar in Figure 1. In this grammar, productions start with lowercase letters whereas terminals start with uppercase (SQL terminal symbols use small caps). Optional statements are delimited by square brackets and alternative sentences are separated by pipes. The grammar defines the following syntactic categories:

- A database `sql_db` is a (non-empty) sequence of relation definitions separated by semicolons (";"). A relation definition assigns a select statement to the relation, that is identified by its name `R` and its schema.
- A schema `sch` is a tuple of attribute names with their corresponding types.
- A select statement `sel_stm` is defined in the usual way. The clauses FROM and WHERE are optional. We also allow UNION and EXCEPT, but notice that the syntax for EXCEPT allows only a relation name instead of a select statement

```
sql_db  ::= R sch := sel_stm;
            ...
            R sch := sel_stm;

sch     ::= (A T,...,A T)

sel_stm ::= SELECT exp,...,exp [ FROM R,...,R [ WHERE wcond ] ]
            | sel_stm UNION sel_stm
            | sel_stm EXCEPT R

exp     ::= C | R.A | exp opm exp | - exp

wcond   ::= TRUE | FALSE | exp opc exp | NOT (wcond)
            | wcond [ AND | OR ] wcond

opm     ::= + | - | / | *

opc     ::= = | <> | < | > | >= | <=
```

`R` stands for relation names, `A` for attribute names, `T` for standard SQL types (as INTEGER, FLOAT, VARCHAR(N)), and `C` for constants belonging to a valid SQL type.

Fig. 1. A Grammar for the R-SQL Language

as usual in SQL. This is done in order to keep simple the syntax and does not imply expressivity losses, because a relation name can be identified with the select statement that defines it.
- An expression `exp` can be either a constant value `C`, an attribute of a relation (denoted by `R.A`), or an arithmetic expression.
- A Boolean condition `wcond` in the WHERE clause of a select statement is built up in the usual way, using also the standard comparison operators.

Below, we show a syntactic transformation $[\_]_{\mathcal{RA}}$ that maps every select statement to an equivalent RA-expression in the usual way[3].

- $[\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m \text{ WHERE } \mathtt{wcond}]_{\mathcal{RA}} = \pi_{\mathtt{exp}_1,\ldots,\mathtt{exp}_k}(\sigma_{\mathtt{wcond}}(\mathtt{R}_1 \times \ldots \times \mathtt{R}_m))$
- $[\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]_{\mathcal{RA}} = [\mathtt{sel\_stm}_1]_{\mathcal{RA}} \bigcup [\mathtt{sel\_stm}_2]_{\mathcal{RA}}$
- $[\mathtt{sel\_stm} \text{ EXCEPT } \mathtt{R}]_{\mathcal{RA}} = [\mathtt{sel\_stm}]_{\mathcal{RA}} - \mathtt{R}$

The formal meaning of every `sel_stm` w.r.t. an interpretation I, stated in Definition 5 (Section 3), evinces the idea that the expected interpretation of a select statement $[\![\mathtt{sel\_stm}]\!]^I$ should be the set of tuples associated to the corresponding equivalent RA-expression $[\mathtt{sel\_stm}]_{\mathcal{RA}}$.

2.2 Expressiveness of R-SQL

Next, we illustrate that R-SQL overcomes some limitations present in current RDBMS's following SQL-99. These languages use NOT EXITS and EXCEPT clauses to deal with negation, and WITH RECURSIVE to engage recursion. As it is pointed out in [5], SQL-99 does not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause. Although any mutual recursion can be converted to direct recursion by inlining [8], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance. For instance, using R-SQL, it is easy to write the classical example for computing even and odd numbers up to a bound (100 in the example) as follows:

```
even(x float) := SELECT 0 UNION
  SELECT odd.x+1 FROM odd WHERE odd.x<100;

odd(x float) := SELECT even.x+1 FROM even WHERE even.x<100;
```

Further, linear recursion in standard SQL restricts the number of allowed recursive calls to be only one, i.e., Fibonacci numbers cannot be specified as follows[4]:

[3] Notice that arithmetic expressions are allowed as arguments in *projection* (π) and *select* (σ) operations.

[4] The relations `fib1` and `fib2` simply represent two aliases for `fib`, which are necessary because, for simplicity, we have not added support for renamings in R-SQL FROM clauses.

```
fib1(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib2(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib(n float, f float) := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
  WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

This means that several graph algorithms specified using non-linear recursion cannot be directly expressed in current recursive SQL systems [17].

Non-termination is another problem that arises associated to recursion. For instance, the basic transitive closure over a graph that includes a cycle makes current SQL systems (such as PostgreSQL and MySQL) either to reject the query or to go into an infinite loop (some systems allow to impose a maximum number of iterations as a simple termination condition). Nevertheless, the fix-point computation used by R-SQL guarantees termination when dealing with finite relations. The following example written in R-SQL defines the relations `arc` (a graph with a cycle) and `path` (its transitive closure). The computation is terminating since both relations are finite.

```
arc(ori varchar(1), des varchar(1)) :=
  SELECT a,b UNION SELECT b,c UNION SELECT c,a;

path(ori varchar(1), des varchar(1)) :=
  SELECT arc.ori, arc.des FROM arc UNION
  SELECT arc.ori, path.des FROM arc,path WHERE arc.des=path.ori
```

The following running example contains a concrete relation defined using the classical transitive closure technique mentioned above.

Example 1. A database for flights. As usual, the information about direct flights can be composed of the city of origin, the city of destination, and the length of the flight. Cities (Lisboa, Madrid, Paris, London, New York) will be represented with constants (`lis`, `mad`, `par`, `lon`, `ny`, resp.)

```
flight(frm varchar(10), to varchar(10), time float) :=
  SELECT 'lis','mad',1.0 UNION SELECT 'mad','par',1.5 UNION
  SELECT 'par','lon',2.0 UNION SELECT 'lon','ny',7.0  UNION
  SELECT 'par','ny',8.0;
```

The relation `reachable` consists of all the possible trips between the cities of the database, maybe concatenating more than one flight.

```
reachable(frm varchar(10), to varchar(10)) :=
  SELECT flight.frm, flight.to FROM flight UNION
  SELECT reachable.frm, flight.to FROM reachable,flight
  WHERE reachable.to = flight.frm;
```

The relation `travel` also gives time information about alternative trips.

```
travel(frm varchar(10), to varchar(10), time float) :=
  SELECT flight.frm, flight.to, flight.time
  FROM flight UNION
  SELECT flight.frm, travel.to, flight.time+travel.time
  FROM flight, travel WHERE flight.to = travel.frm;
```

Both `reachable` and `travel` represent transitive closures of the relation `flight`. Notice that if `flight` has a cycle, then the relation `travel` that includes times for each trip is infinite, while `reachable` is not. As pointed before, `reachable` can be finitely computed in our system. But, as `travel` would produce an infinite set of different tuples, some computation limitation would have to be imposed (as the maximum time for a `travel`, for example). However, this is not a drawback of our approach, but an issue due to using infinite relations (built with arithmetic expressions).

The relation `madAirport` contains travels departing or arriving in Madrid, while `avoidMad` contains possible travels that neither begin, nor end in Madrid.

```
madAirport(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  WHERE (reachable.frm = 'mad' OR reachable.to = 'mad');

avoidMad(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  EXCEPT madAirport;
```

This definition includes negation together with recursive relations. This combination can not be expressed in SQL-99 as it is shown in [4].

3 A Stratified Fixpoint Semantics for R-SQL

It is well-known that the combination of negation and recursion in database languages is a difficult task [1]. This problem has been tackled with stratified fixpoint semantics in several works [12, 11, 14], and we have found that these techniques can be also applied to our proposal to obtain an operational semantics for R-SQL. In this section we present a novel formalization of recursive SQL relations by means of a stratified fixpoint interpretation that formalizes the meaning of R-SQL-databases, and we show how to compute such fixpoint.

Next, we introduce the notions of dependency graph and stratification that provide the basis for the stratified negation formalization we are looking for. Then, we define the concept of stratified interpretations, and prove the existence of the fixpoint of a continuous operator as the required interpretation of a database. The obtained semantics will be the basis of the implementation of a concrete R-SQL database system.

3.1 Dependency Graph and Stratification

Stratification is based on the definition of a *dependency graph* for a database. In the following, we consider a database `sql_db` defined as $\texttt{R}_1\texttt{sch}_1 := \texttt{sel\_stm}_1 ; \ldots ;$

$\texttt{R}_n\texttt{sch}_n$:= $\texttt{sel\_stm}_n$. We denote by RN the set $\{\texttt{R}_1, \ldots, \texttt{R}_n\}$ of relation names of sql_db. We assume that relations are well defined, in the sense that the relation names used inside $\texttt{sel\_stm}_1 \ldots \texttt{sel\_stm}_n$ are in RN. The dependency graph associated to sql_db, denoted by $DG_{\texttt{sql\_db}}$, is a directed graph whose nodes are the elements of RN, and the edges, that can be *negatively labelled*, are determined by the dependencies between the database relations, that are defined as follows. A relation definition of the form `R sch := sel_stm` produces edges in the graph from every relation name inside sel_stm to R. Those edges produced by the relation name that is just to the right of an EXCEPT are negatively labelled.

Definition 1. For every two relations $\texttt{R}_1, \texttt{R}_2 \in \texttt{RN}$, we say:

- $\texttt{R}_2$ *depends* on $\texttt{R}_1$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{sql\_db}}$.
- $\texttt{R}_2$ *negatively depends* on $\texttt{R}_1$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{sql\_db}}$ with at least one negatively labelled edge.

Example 2. Consider the database of Example 1. Its corresponding set of relation names is $\texttt{RN} = \{\texttt{flight}, \texttt{reachable}, \texttt{travel}, \texttt{madAirport}, \texttt{avoidMad}\}$. Its dependency graph is depicted in Figure 2, where negatively labelled edges are annotated with $\neg$.

Definition 2. A *stratification* of sql_db is a mapping $str : \texttt{RN} \to \{1, \ldots, n\}$, such that:

- $str(\texttt{R}_i) \leq str(\texttt{R}_j)$, if $\texttt{R}_j$ depends on $\texttt{R}_i$,
- $str(\texttt{R}_i) < str(\texttt{R}_j)$ if $\texttt{R}_j$ negatively depends on $\texttt{R}_i$.

sql_db is *stratifiable* if there exists a stratification for it. In this case, for every $\texttt{R} \in \texttt{RN}$, we say that $str(\texttt{R})$ is the *stratum* of R. We denote by *numstr* the maximum stratum of the elements of RN. And $str(\texttt{sel\_stm})$ represents the maximum stratum of the relations included in sel_stm.

Intuitively, a relation name preceded by an EXCEPT plays the role of a negated predicate (relation) in the deductive database field. A stratification-based solving procedure ensures that when a relation that contains an EXCEPT in its definition is going to be calculated, the meaning of the inner negated relation has been completely evaluated, avoiding nonmonotonicity, as it is widely studied in Datalog [16]. The novelty lies on introducing these ideas into the field of the relational model.

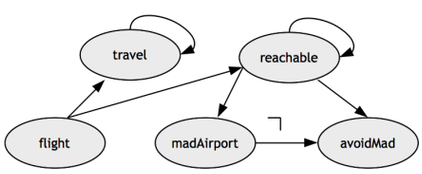


Fig. 2. $DG_{\texttt{sql\_db}}$ of Example 1

3.2 Stratified Interpretations and Fixpoint Operator

From now on, we consider a stratifiable `sql_db`, and that *str* is a stratification for it. In the previous section, we established that in a relation definition for `R sch`, the schema `sch` is a sequence of type declarations for the attributes of `R`. In order to give meaning to this relation, we assume that every type `T` included in `sch` denotes a domain D. In previous examples we have used two types: `varchar`, denoting the domain of strings, and `float`, denoting a numeric domain. We will consider a *universal domain* $\mathcal{D}$ which is the union of the family of the considered domains. Relations of arity k will denote a set of k-tuples included in $\mathcal{D}^k$. In general, every relation denotes a subset of $\mathcal{T} = \bigcup_{n\geq 1} \mathcal{D}^n$.

Interpretations are functions that associate an element of $\mathcal{P}(\mathcal{T})$ to each element of `RN`. So, considering the usual relational model terminology of schema and instance of a relation, the interpretation of a relation in our model can be seen as the relationship between the schema and the instance of the relation. Interpretations are classified by strata. An interpretation of a stratum i gives meaning to the relations of strata less or qual to i. Next, we formalize the concept of interpretation over a stratum.

Definition 3. An *interpretation* I for `sql_db`, over the stratum i, $1 \leq i \leq numstr$ is a function from `RN` to $\mathcal{P}(\mathcal{T})$, such that, for each `R` $\in$ `RN`:

- If `R` has schema $(\mathtt{A}_1\mathtt{T}_1, \ldots, \mathtt{A}_r\mathtt{T}_r)$, and $D_1, \ldots, D_r$ are, respectively, the domains denoted by $\mathtt{T}_1, \ldots, \mathtt{T}_r$, then $I(\mathtt{R}) \subseteq D_1 \times \ldots \times D_r$.
- $I(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

The set of interpretations for `sql_db` over the stratum i, $1 \leq i \leq numstr$ is denoted by $\mathcal{I}_i^{\mathtt{sql\_db}}$. The following definition provides an order on $\mathcal{I}_i^{\mathtt{sql\_db}}$.

Definition 4. Let $i \geq 1$, and $I_1, I_2 \in \mathcal{I}_i^{\mathtt{sql\_db}}$. I_1 *is less or equal than* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if the following conditions are satisfied for every `R` $\in$ `RN`:

- $I_1(\mathtt{R}) = I_2(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$, if $str(\mathtt{R}) = i$.

It is straightforward to check that for any i, $1 \leq i \leq numstr$, $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$ is a poset. The main question is that when an interpretation over a stratum i increases, the set of tuples associated to the relations whose stratum is i can increase, but the sets associated to relations of smaller strata remain invariable. In addition, this poset is a cpo, as it is proved in the following lemma.

Lemma 1. For any $i \geq 1$, the pair $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$ is a complete partially ordered set. Moreover, if $\{I_n\}_{n\geq 0}$ is a chain of interpretations in $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$, then $\hat{I}$, defined as $\hat{I}(\mathtt{R}) = \bigcup_{n\geq 0} I_n(\mathtt{R})$, is the least upper bound of $\{I_n\}_{n\geq 0}$.

Proof. It is easy to prove that $\hat{I} \in \mathcal{I}_i^{\mathtt{sql\_db}}$, and that it is an upper bound. In addition, if I is another upper bound, that implies: If $str(\mathtt{R}) < i$, $I(\mathtt{R}) = I_n(\mathtt{R})$ for every $n \geq 0$, and hence $\hat{I}(\mathtt{R}) = I(\mathtt{R})$. If $str(\mathtt{R}) = i$, $I_n(\mathtt{R}) \subseteq I(\mathtt{R})$ for every $n \geq 0$, then $\bigcup_{n\geq 0} I_n(\mathtt{R}) \subseteq I(\mathtt{R})$. Therefore $\hat{I} \sqsubseteq_i I$, by the definition of $\sqsubseteq_i$. □

The following definition formalizes the meaning of a select statement `sel_stm` in the context of a concrete interpretation I, both associated to a concrete `sql_db` database. As we pointed out before, the interpretation of a `sel_stm` will be the set of tuples associated to its corresponding RA-expression, $[\![\texttt{sel\_stm}]\!]_{\mathcal{RA}}$, when the value of the involved relation names is given by I.

Definition 5. Let $i \geq 1$, and $I \in \mathcal{I}_i^{\texttt{sql\_db}}$. Let `sel_stm` be a select statement including only relation names of `RN`, such that $str(\texttt{sel\_stm}) \leq i$. We recursively define the *interpretation of* `sel_stm` *w.r.t.* I, denoted by $[\![\texttt{sel\_stm}]\!]^I$, as:

- $[\![\texttt{sel\_stm}_1 \text{ UNION } \texttt{sel\_stm}_2]\!]^I = [\![\texttt{sel\_stm}_1]\!]^I \bigcup [\![\texttt{sel\_stm}_2]\!]^I$, where $\bigcup$ stands for the set union.
- $[\![\texttt{sel\_stm} \text{ EXCEPT } \texttt{R}]\!]^I = [\![\texttt{sel\_stm}]\!]^I \setminus I(\texttt{R})$, where $\setminus$ represents set difference.
- $[\![\text{SELECT } \texttt{exp}_1, \ldots, \texttt{exp}_k]\!]^I = \{(exp_1, \ldots, exp_k)\}$, where exp_i is the mathematical evaluation of $\texttt{exp}_i$.
- $[\![\text{SELECT } \texttt{exp}_1, \ldots, \texttt{exp}_k \text{ FROM } \texttt{R}_1, \ldots, \texttt{R}_m \text{ WHERE } \texttt{wcond}]\!]^I =$ $\{(exp_1[\overline{a}/\overline{\texttt{A}}], \ldots, exp_k[\overline{a}/\overline{\texttt{A}}]) | \overline{a} \in I(\texttt{R}_1) \times \ldots \times I(\texttt{R}_m) \text{ and } wcond[\overline{a}/\overline{\texttt{A}}] \text{ is satisfied}\}$.

$\overline{\texttt{A}}$ is a sequence of attributes labelled with their corresponding relation names. More precisely, if $\texttt{A}_1^j, \ldots, \texttt{A}_{r_j}^j$ are the attributes of $\texttt{R}_j$, $1 \leq j \leq m$, then $\overline{\texttt{A}}$ represents the complete sequence $\texttt{R}_1.\texttt{A}_1^1, \ldots, \texttt{R}_1.\texttt{A}_{r_1}^1, \ldots, \texttt{R}_m.\texttt{A}_1^m \ldots \texttt{R}_m.\texttt{A}_{r_m}^m$. $exp_j[\overline{a}/\overline{\texttt{A}}]$, $1 \leq j \leq k$, is the mathematical evaluation of $\texttt{exp}_j$, after replacing the tuple $\overline{a}$ by $\overline{\texttt{A}}$. And $wcond[\overline{a}/\overline{\texttt{A}}]$ is the evaluation of the Boolean expression `wcond`, with the previous substitution.

Example 3. Consider the definitions of the relations `odd` and `even` of Section 2.2. Let us assume a concrete interpretation I such that $I(\texttt{even}) = \{(0), (2)\}$ and $I(\texttt{odd}) = \emptyset$. Hence, the interpretation of the select statement that defines the relation `odd` w.r.t. I is:
$[\![\texttt{SELECT even.x+1 FROM even WHERE even.x<100}]\!]^I = \{(\texttt{even.x+1})[a/\texttt{even.x}]$ $|(a) \in I(\texttt{even})$ and $(\texttt{even.x<100})$ $[a/\texttt{even.x}]$is satisfied$\} = \{(1), (3)\}$.

The case of the relation `even` is analogous:
$[\![\texttt{SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100}]\!]^I =$
$[\![\texttt{SELECT 0}]\!]^I \bigcup [\![\texttt{SELECT odd.x+1 FROM odd WHERE odd.x<100 }]\!]^I = \{\texttt{(0)}\} \bigcup$
$\{(\texttt{odd.x+1})[a/\texttt{odd.x}]$ $|(a) \in I(\texttt{odd}), (\texttt{odd.x<100})[a/\texttt{odd.x}]$ is satisfied$\} = \{\texttt{(0)}\}$.

Notice that the interpretation $\hat{I}$ defined by $\hat{I}(\texttt{even}) = \{(0), (2), \ldots, (100)\}$ and $\hat{I}(\texttt{odd}) = \{(1), (3), \ldots, (99)\}$ satisfies:
$\hat{I}(\texttt{even}) = [\![\texttt{SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100}]\!]^{\hat{I}}$.
$\hat{I}(\texttt{odd}) = [\![\texttt{SELECT even.x+1 FROM even WHERE even.x<100}]\!]^{\hat{I}}$.

The semantics of `sql_db` will be formalized by means of an interpretation I over *numstr*, such that for every $\texttt{R} \in \texttt{RN}$, if `R sch := sel_stm` is the definition of `R` in `sql_db`, then I maps the set $[\![\texttt{sel\_stm}]\!]^I$ to `R`, as the interpretation $\hat{I}$ of Example 3 does. For every stratum i, the appropriate interpretation that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator over the set of interpretations of stratum i. These fixpoint interpretations are constructed sequentially from stratum 1 to *numstr*.

The fixpoint of the last stratum *numstr* provides the semantics for the whole database. Some technical lemmas are shown in order to ensure the existence of such fixpoint interpretations.

The following lemma states that the sets of tuples denoted by a select statement of a stratum i, w.r.t. two ordered interpretations, satisfy an inclusion relation that is in accordance with the order $\sqsubseteq_i$ between the two interpretations.

Lemma 2. Let $i \geq 1$, $\mathtt{R} \in \mathtt{RN}$, with $str(\mathtt{R}) \leq i$, and I_1, $I_2 \in \mathcal{I}_i^{\mathtt{sql\_db}}$, such that $I_1 \sqsubseteq_i I_2$. Then, every `sel_stm` included in the select statement that defines `R` holds:

- If $str(\mathtt{sel\_stm}) < i$, then $[\![\mathtt{sel\_stm}]\!]^{I_1} = [\![\mathtt{sel\_stm}]\!]^{I_2}$.
- If $str(\mathtt{sel\_stm}) = i$, then $[\![\mathtt{sel\_stm}]\!]^{I_1} \subseteq [\![\mathtt{sel\_stm}]\!]^{I_2}$.

Proof. The proof is inductive on the structure of `sel_stm`. Here, we only show the most critical case. The others are similar.

$[\![\mathtt{sel\_stm}$ EXCEPT $\mathtt{R'}]\!]^{I_1} = [\![\mathtt{sel\_stm}]\!]^{I_1} \setminus I_1(\mathtt{R'})$. According to the definition of stratification, $str(\mathtt{R'}) < i$, because we are assuming that `sel_stm` EXCEPT $\mathtt{R'}$ occurs in the definition of `R` and $str(\mathtt{R}) \leq i$. Hence $I_1(\mathtt{R'}) = I_2(\mathtt{R'})$. Now, if $str(\mathtt{sel\_stm}$ EXCEPT $\mathtt{R'}) \leq i$, then $[\![\mathtt{sel\_stm}]\!]^{I_1} \subseteq [\![\mathtt{sel\_stm}]\!]^{I_2}$, applying the induction hypothesis. Therefore $[\![\mathtt{sel\_stm}$ EXCEPT $\mathtt{R'}]\!]^{I_1} \subseteq [\![\mathtt{sel\_stm}$ EXCEPT $\mathtt{R'}]\!]^{I_2}$, with equality for the case $str(\mathtt{sel\_stm}$ EXCEPT $\mathtt{R'}) < i$. □

The following lemma underlies the proof of the continuity of the operator whose fixpoint provides the semantics of a database (it can be proved by induction on the structure of `sel_stm`).

Lemma 3. Let $i \geq 1$, $\mathtt{R} \in \mathtt{RN}$, with $str(\mathtt{R}) \leq i$, and $\{I_n\}_{n\geq 0}$ be a chain in $\mathcal{I}_i^{\mathtt{sql\_db}}$. Then, for every `sel_stm` included in the definitions of `R`, if $\hat{I} = \bigsqcup_{n\geq 0} I_n$, there exists $n \geq 0$, such that $[\![\mathtt{sel\_stm}]\!]^{\hat{I}} = [\![\mathtt{sel\_stm}]\!]^{I_n}$.

Next, for every i, a continuous operator T_i over the set $\mathcal{I}_i^{\mathtt{sql\_db}}$ of interpretations of stratum i is defined. Analogously to the theoretical foundations that support Datalog [16], we choose the least fixpoint of T_i, as the interpretation over i that will give meaning to the relations of stratum i. In accordance with the Knaster-Tarski theorem, this fixpoint can be obtained as the least upper bound of the chain of interpretations resulting by successively applying this operator to a minimal interpretation.

Definition 6. Let $1 \leq i \leq numstr$. The *operator* $T_i : \mathcal{I}_i^{\mathtt{sql\_db}} \longrightarrow \mathcal{I}_i^{\mathtt{sql\_db}}$ transforms interpretations over i as follows. For every $I \in \mathcal{I}_i^{\mathtt{sql\_db}}$, $\mathtt{R} \in \mathtt{RN}$:

- $T_i(I)(\mathtt{R}) = I(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $T_i(I)(\mathtt{R}) = [\![\mathtt{sel\_stm}]\!]^I$, if $str(\mathtt{R}) = i$ and `R sch := sel_stm` is the definition of `R` in `sql_db`.
- $T_i(I)(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

This operator is proved to be monotone (it is a consequence of Lemma 2) and continuous for every i.

Lemma 4. [Monotonicity of T_i] Let $i \geq 1$ and $I_1, I_2 \in \mathcal{I}_i^{\texttt{sql\_db}}$, such that $I_1 \sqsubseteq_i I_2$. Then, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.

Proposition 1. [Continuity of T_i] Let $i \geq 1$ and $\{I_n\}_{n\geq 0}$ be a chain of interpretations in $\mathcal{I}_i^{\texttt{sql\_db}}$ ($I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$). Then, $T_i(\bigsqcup_{n\geq 0} I_n) =_i \bigsqcup_{n\geq 0} T_i(I_n)$.

Proof. The proof of $\bigsqcup_{n\geq 0} T_i(I_n) \sqsubseteq_i T_i(\bigsqcup_{n\geq 0} I_n)$ is a direct consequence of the monotonicity of T_i (Lemma 4). Let us prove $T_i(\bigsqcup_{n\geq 0} I_n) \sqsubseteq_i \bigsqcup_{n\geq 0} T_i(I_n)$:

- If $str(\texttt{R}) < i$, then $T_i(\bigsqcup_{n\geq 0} I_n)(\texttt{R}) = \bigsqcup_{n\geq 0} I_n(\texttt{R})$, by the definition of T_i. Now, for every $n \geq 0$, $I_n(\texttt{R}) = T_i(I_n)(\texttt{R})$, also by definition of T_i. Therefore, $(T_i(\bigsqcup_{n\geq 0} I_n))(\texttt{R}) = (\bigsqcup_{n\geq 0} T_i(I_n))(\texttt{R})$.
- If $str(\texttt{R}) = i$, then $T_i(\bigsqcup_{n\geq 0} I_n)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{\bigsqcup_{n\geq 0} I_n}$, by definition of T_i. And, in accordance with Lemma 3, for some $n \geq 0$: $[\![\texttt{sel\_stm}]\!]^{\bigsqcup_{n\geq 0} I_n} \subseteq [\![\texttt{sel\_stm}]\!]^{I_n}$. Now $[\![\texttt{sel\_stm}]\!]^{I_n} = T_i(I_n)(\texttt{R})$, by definition of T_i, and obviously $T_i(I_n)(\texttt{R}) \subseteq \bigcup_{n\geq 0} T_i(I_n)(\texttt{R})$, but $\bigcup_{n\geq 0} T_i(I_n)(\texttt{R}) = (\bigsqcup_{n\geq 0} T_i(I_n))(\texttt{R})$, by Lemma 1. Hence, we conclude $T_i(\bigsqcup_{n\geq 0} I_n)(\texttt{R}) \subseteq (\bigsqcup_{n\geq 0} T_i(I_n))(\texttt{R})$. □

Next, the expected result corresponding to the existence of least fixpoint stratum by stratum is shown.

Lemma 5. The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$, where $\emptyset : \texttt{RN} \to \mathcal{P}(\mathcal{T})$ is the interpretation such that $\emptyset(\texttt{R}) = \emptyset$ for every $\texttt{R} \in \texttt{RN}$.

Proof. By the Knaster-Tarski fixpoint theorem [15], using Proposition 1. □

We will denote $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$ by fix_1, i.e., fix_1 represents the least fixpoint at stratum 1. Using Example 1, Figure 3 shows the tuples corresponding to the successive applications of the operator T_1 until $fix_1(\texttt{travel})$ is obtained.

Consider now the sequence $\{T_2^n(fix_1)\}_{n\geq 0}$ of interpretations in $(\mathcal{I}_2^{\texttt{sql\_db}}, \sqsubseteq_2)$ greater than fix_1. Using the definition of T_i and the fact that $fix_1(\texttt{R}) = \emptyset$ for every R such that $str(\texttt{R}) \geq 2$, it is easy to prove, by induction on $n \geq 0$, that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2(T_2(fix_1)) \sqsubseteq_2 \ldots, \sqsubseteq_2 T_2^n(fix_1), \ldots$$

$T_1^n(\emptyset)(\texttt{travel})$	Set of tuples
$T_1^1(\emptyset)(\texttt{travel})$	{(lon,ny,7.0), (par,lon,2.0), (par,ny,8.0), (mad,par,1.5), (lis,mad,1.0)}
$T_1^2(\emptyset)(\texttt{travel})$	{(lis,par,2.5), (par,ny,9.0), (mad,ny,9.5), (mad,lon,3.5)}
$T_1^3(\emptyset)(\texttt{travel})$	{(lis,ny,10.5), (lis,lon,4.5), (mad,ny,10.5)}
$T_1^4(\emptyset)(\texttt{travel})$	{(lis,lon,4.5), (mad,ny,10.5), (lis,ny,11.5)}

Fig. 3. Obtaining $fix_1(\texttt{travel})$

As before, in accordance with Proposition 1, $\{T_2^n(fix_1)\}_{n\geq 0}$ has a least upper bound, $\bigsqcup_{n\geq 0} T_2^n(fix_1)$, in $(\mathcal{I}_2^{\texttt{sql\_db}}, \sqsubseteq_2)$ that is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2.

By proceeding successively, for every i, $1 < i \leq numstr$, a chain:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots \sqsubseteq_i T_i^n(fix_{i-1}) \ldots$$

can be defined, and a fixpoint of T_i, $fix_i = \bigsqcup_{n\geq 0} T_i^n(fix_{i-1})$, can be found.

Theorem 1. There is a fixpoint interpretation $fix : \texttt{RN} \longrightarrow \mathcal{P}(\mathcal{T})$, such that for every $\texttt{R} \in \texttt{RN}$, if `sel_stm` is the definition of R, then $fix(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix}$.

Proof. The interpretation fix we are looking for is fix_{numstr}, the least fixpoint of the operator T_{numstr}, applied to $fix_{numstr-1}$. As it has been pointed out, this fixpoint exists and verifies $fix_1 \sqsubseteq_{numstr} fix_2 \sqsubseteq_{numstr} \ldots \sqsubseteq_{numstr} fix_{numstr}$. Moreover, if $str(\texttt{R}) = i$, $1 \leq i \leq numstr$, and it is defined by the statement `sel_stm`, then $fix(\texttt{R}) = fix_i(\texttt{R}) = T_i(fix_i)(\texttt{R})$, because fix_i is the fixpoint of T_i. Now, $T_i(fix_i)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix_i}$, by definition of T_i. We can conclude $fix(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix}$, trivially if $i = numstr$, or using Lemma 2, if $i < numstr$, because $fix_i \sqsubseteq_{numstr} fix$. □

Therefore, the interpretation fix defines the fixpoint semantics of `sql_db`. This semantics is the support of the database system prototype we have implemented, which is described next.

4 Implementing R-SQL

In this section we introduce a working proof-of-concept implementation for the R-SQL language that takes a set of relation definitions and outputs their meanings if a stratification can be found. More specifically, taking a stratifiable database definition in the R-SQL syntax as input, we get a SQL database (for a concrete SQL database system), that corresponds to the fixpoint semantics of the input database. If the database is not stratifiable, the system throws an error message and stops.

4.1 An Algorithm to Compute the Database Fixpoint

Let `sql_db` be the definition of a R-SQL database. In order to create the corresponding SQL database we have to generate the appropriate SQL sentences for building the expected relations, that will be eventually processed by a RDBMS. The algorithm takes `sql_db` as input, i.e., a sequence of relation definitions, $\texttt{R}_1\texttt{sch}_1$:= `sel_stm`$_1$; ...; $\texttt{R}_n\texttt{sch}_n$:= `sel_stm`$_n$. The computation builds the dependency graph for `sql_db`, as shown in Section 3.1, then calculates a stratification for it obtaining the sets $\mathrm{R}_1, \ldots, \mathrm{R}_{numstr}$, where R_i is the set of relations of stratum i, and finally the fixpoint is computed with the following algorithm:

```
(1)   str:=1
(2)   while str ≤ numstr do
(3)       for each Rᵢ ∈ R_str do CREATE TABLE Rᵢ schᵢ
(4)       change := true
(5)       while change do
(6)           size := rel_size_sum(R_str)
(7)           for each Rᵢ ∈ R_str do
(8)               INSERT INTO Rᵢ SELECT * FROM sel_stmᵢ
(9)                   EXCEPT SELECT * FROM Rᵢ;
(10)          change = (size ≠ rel_size_sum(R_str))
(11)      end while
(12)      str:=str+1
(13)  end while
```

This algorithm generates for each $\mathtt{R}_i$ of `sql_db` a SQL table with the elements of $fix(\mathtt{R}_i)$. Each iteration of the external *while* at line 2 corresponds to a stratum *str*, and builds the tables of the relations of this stratum, by calculating fix_{str}. To do that, first of all an empty table with the corresponding attributes is created for every relation in the stratum *str* (line 3). Then, the iteration n of the innermost *while* at line 5 computes $T^n_{str}(fix_{str-1})$, as we will explain. For every relation $\mathtt{R}_i$ of *str*, it submits the INSERT statement at line 8. The sentence SELECT * FROM $\mathtt{sel\_stm}_i$ selects all tuples as defined by the relation $\mathtt{R}_i$ (notice that $\mathtt{sel\_stm}_i$ is a valid SQL statement). Assuming that the current database instance coincides with the value of the interpretation $T^{n-1}_{str}(fix_{str-1})$, then in accordance with Definition 5, the set of tuples that satisfy that SQL statement coincides with $[\![\mathtt{sel\_stm}_i]\!]^{T^{n-1}_{str}(fix_{str-1})}$. And this is $T^n_{str}(fix_{str-1})(\mathtt{R}_i)$, by Definition 6. The tuples already present in the table are excluded to avoid repetitions (with the EXCEPT clause at line 9). In this way, $T^n_{str}(fix_{str-1})(\mathtt{R}_i)$ is obtained for every $\mathtt{R}_i$ of stratum *str*. The expression $rel\_size\_sum(\mathrm{R}_{str})$ at line 10 is equal to $\sum_{\mathtt{R}\in R_{str}} |\mathtt{R}|$, where $|\mathtt{R}|$ is the current number of tuples of the table corresponding to R. Therefore, the variable *change* controls changes on the table sizes in order to stop the process, since *change* = *false* means that $T^n_{str}(fix_{str-1}) = T^{n-1}_{str}(fix_{str-1})$, so that fix_{str} has been reached. Then, the last iteration of the external *while* calculates fix_{numstr}, the fixpoint of `sql_db`.

4.2 A Concrete Implementation

The concrete implementation of this algorithm can be done in a number of ways. We have developed a Prolog program that processes the R-SQL input file, builds the dependency graph and the stratification (if exists), and finally produces a Python module with the code of the previous section. In fact, the external *while* at line 2 is expanded according to the number of strata, writing explicitly the corresponding code for each stratum. The *for* loop at line 7 is also expanded as we will see in Example 4. We have chosen Python as the host language mainly because is multiplatform and it provides easy connections with different database

systems such as PostgreSQL, MySQL, or even via ODBC, which allows connectivity to almost any RDBMS. The additional features required for the host language are basic: Loops, assignment and basic arithmetic.

Example 4. Below, we show the result of executing our proposed algorithm for the `sql_db` of Example 1. The system assigns stratum 1 to `flight, reachable, travel, madAirport`, and stratum 2 to `avoidMad`. Next, we detail some parts of the code generated stratum by stratum. Firstly, for stratum 1, we have:

while change **do**
size := rel_size_sum(R_{str})
```
INSERT INTO flight SELECT 'lis','mad',1 UNION SELECT 'mad','par',1
UNION SELECT 'par','lon',2 UNION SELECT 'lon','ny',7
UNION SELECT 'par','ny',8 EXCEPT SELECT * FROM flight;

INSERT INTO reachable SELECT flight.frm, flight.to
FROM flight UNION SELECT reachable.frm, flight.to
WHERE reachable.to = flight.frm
EXCEPT SELECT * FROM reachable;

INSERT INTO travel SELECT * FROM flight UNION
SELECT flight.frm, travel.to, flight.time+travel.time
FROM flight, travel WHERE flight.to = travel.frm
EXCEPT SELECT * FROM travel;

INSERT INTO madAirport SELECT travel.frm,travel.to
FROM travel EXCEPT SELECT * FROM madAirport;
```
change = (size $\neq$ rel_size_sum(R_{str}))
end while

In the first iteration of this loop, we obtain all the tuples for `flight` and `madAirport` relations. But the recursive definitions for `reachable` and `travel` need more iterations. As mentioned before, those iterations correspond to the successive applications of T_1. The tuples added for `travel` at each iteration are shown in Figure 3 (Section 3.2). After five iterations, the loop stops and the first stratum is completed. In the second stratum we consider the `avoidMad` relation:

```
INSERT INTO avoidMad SELECT travel.frm,travel.to FROM travel
EXCEPT SELECT * FROM madAirport EXCEPT SELECT * FROM avoidMad;
```

This second loop ends after two iterations. This completes fix_2 for our `sql_db`, i.e., it obtains the semantics of the working example database.

4.3 Integrating R-SQL into a RDBMS

Our proposal establishes the core for introducing a novel approach for recursion in SQL. The current implementation of R-SQL has been conceived as a proof-of-concept of the theoretical foundations of the language. As we have stated, this leads to compute the semantics of the whole database from scratch. Nevertheless, the main goal of the proposal is not to introduce a new database language, but

to allow less-restricted recursive relation definitions in existing SQL engines. In that sense, our proposal can be understood as the foundation of an existing SQL RDBMS that supports extended forms of recursion, allowing users to define recursive relations as regular views using the R-SQL techniques, developed in this work. Once an R-SQL database definition has been processed, the tables obtained can be stored as a database instance in a concrete RDBMS. On the one hand, the user can formulate queries that will be solved using those tables (without performing any further fixpoint computation). On the other hand, as we pointed out before, the user can define new recursive relations using views. Those views can be readily used in conjunction with other regular views, and they can be either computed on demand or can be materialized.

In order to compute the answer of new recursive relations, the current (relation) instance can be considered as a stratified R-SQL database. It is correct to assign higher strata to the new relations, because none of the existing relations depend on the new ones, and a relation definition does not introduce dependencies between the relations that appear in its select statement. Then, their tuples can be obtained by executing the algorithm in Section 4.1 to compute the fixpoint of their corresponding strata, therefore saving recalculating the previous ones. Moreover, it is straightforward to modify the algorithm to get a lazy evaluation of such relations, performing iterations only when new values are demanded. To seamlessly integrate this into a RDBMS, we can profit from the fourth-generation languages (e.g., SQL PL in IBM DB2 and PL/SQL in Oracle).

5 Conclusions

In this paper, we have introduced the R-SQL language as a new approach for incorporating recursion in SQL. This is not a trivial task, and it was not addressed in the initial proposals of SQL. It was firstly introduced in the 1999 standard, allowing only a limited form of recursion, namely linear recursion, which does not allow neither multiple recursive calls nor mutually recursive definitions. The difficulties increase when recursion is combined with negation.

We have developed a theoretical framework and a suitable implementation for R-SQL, inspired on the stratification techniques and fixpoint computations used for instance in Datalog. The stratification mechanism implies to impose some syntactic conditions on the database definitions, that guarantee that the fixpoint for such a database can be computed in a finite number of steps. This condition is less restrictive than the linearity conditions required by the standard SQL. This means that our approach is more expressive than the one adopted in SQL; in addition our language is supported by a solid computational semantics. We have presented a proof-of-concept implementation of the R-SQL database definition language based on this semantics. This implementation produces as output a set of standard SQL statements embedded in a Python program that builds the relational tables corresponding to the fixpoint of the input database definition. This implementation has been tested with PostgreSQL, but the architecture can

be easily ported to any RDBMS. The system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQL`.

As already suggested, our approach can be integrated into a state-of-the-art RDBMS. This can be dealt by resorting to database function definitions, which allow cursor-returning functions. In addition for this integration to be practical, performance improvements play a key role as, e.g., indexing of temporary relations during fixpoint computations and identifying tuple seeds in relation definitions that do not need to be recomputed.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):377–390, June 1970.
3. C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.
4. S. J. Finkelstein, N. Mattos, I. S. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. Technical report, ISO, 1996.
5. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.
6. M. A. W. Houtsma and P. M. G. Apers. Algebraic optimization of recursive queries. *Data Knowl. Eng.*, 7:299–325, 1991.
7. ISO/IEC. SQL:2008 ISO/IEC 9075(1-4,9-11,13,14):2008 Standard, 2008.
8. O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.*, 2(1-4):151–164, Mar. 1993.
9. R. A. Kowalski. Logic for data description. In *Logic and Data Bases*, pages 77–103, 1977.
10. I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Rec.*, 23:103–114, May 1994.
11. S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08*, volume 4989 of *LNCS*, pages 289–304. Springer-Verlag, 2008.
12. K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122, 1994.
13. R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, pages 191–233, 1982.
14. J. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Kaufmann, Los Altos, CA, 1988.
15. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
16. J. Ullman. *Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.
17. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., 1997.

Incorporating Hypothetical Views and Extended Recursion into SQL Database Systems *

Gabriel Aranda-López[1], Susana Nieva[1],
Fernando Sáenz-Pérez[2] and Jaime Sánchez-Hernández[1]

[1] Dept. Sistemas Informáticos y Computación, UCM, Spain
[2] Dept. Ingeniería del Software e Inteligencia Artificial, UCM, Spain
garanda@fdi.ucm.es, {nieva,fernan,jaime}@sip.ucm.es

Abstract

Current database systems supporting recursive SQL impose restrictions on queries such as linearity, and do not implement mutual recursion. In a previous work we presented the language and prototype R-SQL to overcome those drawbacks. Now we introduce a formalization and an implementation of the database system HR-SQL that, in addition to extended recursion, incorporates hypothetical reasoning in a novel way which cannot be found in any other SQL system, allowing both positive and negative assumptions. The formalization extends the fixpoint semantics of R-SQL. The implementation improves the efficiency of the previous prototype and is integrated in a commercial DBMS.

1 Introduction

Current relational database systems provide limited support for the ANSI/ISO standard language SQL w.r.t. recursion. In [2] we proposed a new approach, called R-SQL, aimed to overcome some of such limits. We developed a formal framework, borrowing techniques from the deductive database field, such as stratified negation [15], and following the original relational data model [7], so avoiding both duplicates and nulls (as encouraged by [8]). But in addition to recursion, several applications require predictive and historical analysis over large amounts of data [10], typically making some sort of assumptions to deduce conclusions. Hypothetical queries, also known as "what-if" queries, can help managers to take decisions on scenarios that are somewhat changed with respect to a current state. Such queries are used, for instance, for deciding what resources must be added, changed or removed to optimize some criterion. Current applications include OLAP environments business intelligence, and e-commerce.

So, driven by these needs, with the work proposed in this paper we face the inclusion of hypothetical queries and views in the recursive SQL setting based on [2]. To this end, we extend a subset of standard SQL to embody both recursive definitions and hypothetical views in the language HR-SQL. We summarize the syntax and semantics of the definition language in Section 2, and introduce a novel syntax and semantics of queries and view definitions in sections 3 and 4, respectively. An assumption (hypothetical reasoning) can be either overloading the relation (with the new clause ASSUME `query` IN `relation`) or restricting it (ASSUME `query` NOT IN `relation`). For supporting our approach, we propose a stratified fixpoint semantics which is an extension of the semantics presented in [2] to give meaning to hypothetical queries and view definitions.

Since our targets are current state-of-the-art relational database systems (DBMS's), we adhere to stratification in order to return a single answer set [15], which is a natural expectation

*This work has been partially supported by the Spanish projects S2009/TIC-1465 (PROMETIDOS) and UCM-BSCH-GR58/08-910502 (GPD-UCM).

from current database users. In Section 5, we propose an implementation of the HR-SQL language for the concrete system IBM DB2 (although it is easily adaptable to any other system), which improves the prototype introduced in [2] by factoring out those fragments of SQL queries that can be computed out of the fixpoint operator loop. In addition, we propose an efficient query solving procedure by generating SQL PL scripts and temporary tables to avoid locks and logging, therefore providing memory scalability and performance. Moreover, we provide a shell in which users can submit regular, hypothetical, and extended recursive SQL queries. Whereas regular queries are directly sent to the host DBMS, hypothetical and recursive queries are processed by such SQL PL scripts.

Related Work. To the best of our knowledge, there have been neither a formalization nor a system for SQL combining recursion and hypothetical queries as we do. However, we list some related works in both the relational and logic programming fields. With respect to hypothetical relational databases, the very first work was presented in [13], where hypotheses were stated by replacing actual data with a REPLACE operator, and assumed data persist until the query is finished. In that early work, recursion was not considered. Works as [9] present extensions of RA to support hypothetical queries by means of updates and with no recursion. Also, the educational system DES [12] includes hypothetical SQL queries, but neither hypothetical views nor negative assumptions are supported. On the logic programming arena, Hypothetical Datalog [3, 5] fits into intuitionistic logic programming, an extension of logic programming including both embedded implications and negation, and integrates atomic assumptions as hypothetical queries in the inference system. It has been a proposal thoroughly studied from semantic and complexity point-of-views, allowing to assume atoms in order to prove goals. Transaction logic [4] allows a database to be updated by transactions with elementary updates, and the transaction base is immutable. If bulk updates are needed, the transaction base must account for them. In [1] we developed a more expressive setting for constraint deductive databases based on Hereditary Harrop formulas. In particular, it provides support for assuming rules as hypothetical queries. Our current work can be understood as porting this feature to relational databases by adding the assume clause involving assumptions over relations which intensionally add new tuples (i.e., with select statements) to such relations. As a surplus, in this work, we allow to intensionally remove tuples from such relations by negative assumptions.

2 The Definition Database Language of HR-SQL

This language is oriented to provide definition for databases using a SQL-like language, which allows to define recursive definitions of relations. The formal syntax for a database definition is described by the following grammar:

```
db       ::=   R sch := sel_stm; ... R sch := sel_stm;

sch      ::=   (A T,...,A T)

sel_stm  ::=   SELECT exp,...,exp [FROM R,...,R [WHERE cond]]
               | sel_stm UNION sel_stm  | sel_stm EXCEPT sel_stm

exp      ::=   C | R.A | exp m_op exp | -exp

cond     ::=   TRUE | FALSE | exp b_op exp | NOT cond | cond [AND|OR] cond

m_op     ::=   + | - | / | *

b_op     ::=   = | <> | < | > | >= | <=
```

Uppercase identifiers denote terminal symbols and lowercase ones denote grammar productions, `R` stands for relation names, `A` for attribute names, `T` for standard SQL types (as INTEGER, FLOAT, VARCHAR(N)), `cond` for Boolean conditions, `m_op` and `b_op` for mathematical and Boolean operators respectively, and `C` for constants of a valid SQL type.

A database `db` is a (non-empty) sequence of relation definitions. A relation definition assigns a select statement to the relation, that is identified by its name `R` and its schema `sch`, that is a tuple of attribute names with their corresponding types. As syntactic sugar, we admit `*` in the projection list of SQL statements. The HR-SQL definition language coincides with the R-SQL introduced in [2], but the syntax of the EXCEPT operator allows now any select statement in the right part, instead of a simple relation name (as it was the case in [2]).

Example 1. *The travel database definition below (inspired on an example of [6]) represents the information sketched in Figure 1. This database includes the relations* `flight`, `bus` *and* `boat` *with schema* `(ori varchar(10), des varchar(10), time float)` *to store information about origin (*`ori`*), destination (*`des`*) and time (*`time`*), for traveling around the Canary Islands.*

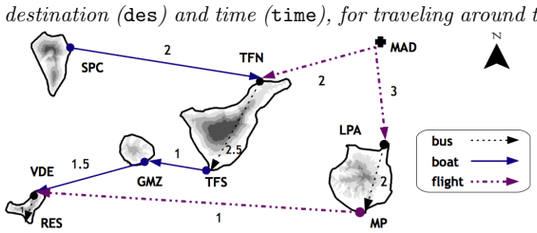


Figure 1: Travel Database for the Canary Islands.

The relation `link` *collects all the possible transports. The relation* `travel` *is the transitive closure of* `link`*, i.e., it provides all the possible travels of the database, maybe concatenating any of the available transports. Their respective definitions written in HR-SQL syntax are:*

```
link(ori varchar(10), des varchar(10), time float):=
   SELECT * FROM flight UNION SELECT * FROM boat UNION
   SELECT * FROM bus;
travel(ori varchar(10), des varchar(10), time float):=
   SELECT * FROM link UNION
   SELECT link.ori, travel.des, link.time + travel.time
          FROM link, travel WHERE link.des=travel.ori;
```

From now on, $\mathtt{RN}_{\mathtt{db}}$ stands for the set of relations names $\{\mathtt{R}_1, \ldots, \mathtt{R}_n\}$ defined in a database `db`. We write $\mathtt{RN}_{\mathtt{sel\_stm}}$ for the set of relation names occurring in a select statement `sel_stm`. For the case of a select statement of the form $\mathtt{sel\_stm} = \mathtt{sel\_stm}_1$ EXCEPT $\mathtt{sel\_stm}_2$ we also define $\mathtt{RN}^{\neg}_{\mathtt{sel\_stm}}$ as the set of relation names occurring in $\mathtt{sel\_stm}_2$ (notice that $\mathtt{RN}^{\neg}_{\mathtt{sel\_stm}} \subseteq \mathtt{RN}_{\mathtt{sel\_stm}}$). We assume that for every `R sch:= sel_stm` defined in `db` it holds that $\mathtt{RN}_{\mathtt{sel\_stm}} \subseteq \mathtt{RN}_{\mathtt{db}}$.

2.1 Fixpoint Semantics

The meaning of every relation defined in a database `db` corresponds to the set of tuples that "satisfies" the relation definition. In [2] a stratified fixpoint semantics for the language R-SQL was introduced. Here, we recapitulate the main concepts in order to facilitate the understanding of the following sections. In addition, we introduce the semantics of the extended EXCEPT select statement.

The stratified fixpoint theory holds on the notion of dependency graph for a database. The *dependency graph* associated to db, denoted by $DG_{\mathtt{db}}$, is a directed graph whose nodes are the elements of $\mathtt{RN}_{\mathtt{db}}$, and the edges (which can be negatively labeled) are determined as follows. For any relation definition `R sch := sel_stm` there is an edge from every relation name $\mathtt{R}' \in \mathtt{RN}_{\mathtt{sel\_stm}}$ to R. Those edges produced by the relation names belonging to $\mathtt{RN}^{\neg}_{\mathtt{sel\_stm}}$ are *negatively labeled*. Then, for every pair of relations $\mathtt{R}_1, \mathtt{R}_2 \in \mathtt{RN}_{\mathtt{db}}$, we say that $\mathtt{R}_2$ *depends* on $\mathtt{R}_1$ if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$. And $\mathtt{R}_2$ *negatively depends* on $\mathtt{R}_1$ if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$ with at least one negatively labeled edge. The previous concepts are needed to characterize the stratifiable databases.

Definition 1. *A* stratification *of a database* db *defining n relations is a mapping* $str : \mathtt{RN}_{\mathtt{db}} \to \{1, \ldots, n\}$*, such that:* $str(\mathtt{R}_i) \leq str(\mathtt{R}_j)$*, if* $\mathtt{R}_j$ *depends on* $\mathtt{R}_i$*, and* $str(\mathtt{R}_i) < str(\mathtt{R}_j)$*, if* $\mathtt{R}_j$ *negatively depends on* $\mathtt{R}_i$*.*

The database db *is* stratifiable *if there exists a stratification for it. In this case, for every* $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$*, we say that* $str(\mathtt{R})$ *is the* stratum *of* R*. And for a select statement* sel_stm*, we define* $str(\mathtt{sel\_stm}) = max\{str(\mathtt{R}_i) \mid \mathtt{R}_i \in \mathtt{RN}_{\mathtt{sel\_stm}}\}$*.*

From now on, we consider a fixed stratifiable database db and a stratification *str* for it. In order to give meaning to a relation $\mathtt{R}\ (\mathtt{A}_1\ \mathtt{T}_1, ..., \mathtt{A}_r\ \mathtt{T}_r)$, we assume that every type $\mathtt{T}_i$, $i = 1..r$, denotes a domain D_i. We also assume a *universal domain* $\mathcal{D}$, which is the union of the family of the considered domains. Since different relations can have different arities, we use the set $\mathcal{T} = \bigcup_{n \geq 1} \mathcal{D}^n$. Interpretations are defined as functions that associate an element of $\mathcal{P}(\mathcal{T})$ to each element of $\mathtt{RN}_{\mathtt{db}}$, and they are classified by strata, as we formalize next.

Definition 2. *Let* $i \geq 1$*, an* interpretation I *for* db*, over the stratum* i*, is a function* $I : \mathtt{RN}_{\mathtt{db}} \to \mathcal{P}(\mathcal{T})$*, such that for every* $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$ *with schema* sch*:*

- *If* $\mathtt{sch} \equiv (\mathtt{A}_1\ \mathtt{T}_1, \ldots, \mathtt{A}_r\ \mathtt{T}_r)$*, and* $D_1, \ldots, D_r$ *are, respectively, the domains denoted by* $\mathtt{T}_1, \ldots, \mathtt{T}_r$*, then* $I(\mathtt{R}) \subseteq D_1 \times \ldots \times D_r$*,*
- $I(\mathtt{R}) = \emptyset$*, if* $str(\mathtt{R}) > i$*.*

The set of interpretations for db *over the stratum* i *is denoted by* $\mathcal{I}_i^{\mathtt{db}}$*. Let* $I_1, I_2 \in \mathcal{I}_i^{\mathtt{db}}$*.* I_1 is less than or equal to I_2 at stratum i*, denoted by* $I_1 \sqsubseteq_i I_2$*, if the following conditions are satisfied for every* $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$*:* $I_1(\mathtt{R}) = I_2(\mathtt{R})$*, if* $str(\mathtt{R}) < i$*, and* $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$*, if* $str(\mathtt{R}) = i$*.*

It is straightforward to check that for any i, $(\mathcal{I}_i^{\mathtt{db}}, \sqsubseteq_i)$ is a poset. The main question is that when an interpretation over a stratum i increases, the set of tuples associated to the relations whose stratum is i can increase, but the sets associated to relations of smaller strata remain invariable. In addition, $(\mathcal{I}_i^{\mathtt{db}}, \sqsubseteq_i)$ is a complete partially ordered set: If $\{I_n\}_{n \geq 0}$ is a chain in $(\mathcal{I}_i^{\mathtt{db}}, \sqsubseteq_i)$, then $\hat{I}$, defined as $\hat{I}(\mathtt{R}) = \bigcup_{n \geq 0} I_n(\mathtt{R})$, $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$, is the least upper bound of $\{I_n\}_{n \geq 0}$.

The following definition formalizes the meaning of a select statement sel_stm in the context of a concrete interpretation I.

Definition 3. *Let* $i \geq 1$*,* $I \in \mathcal{I}_i^{\mathtt{db}}$*. Let* sel_stm *be a select statement, such that* $str(\mathtt{sel\_stm}) \leq i$*. We recursively define the* interpretation of sel_stm w.r.t. I *for* db*, denoted by* $[\![\mathtt{sel\_stm}]\!]^I$*, as follows:*

- $[\![\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]\!]^I = [\![\mathtt{sel\_stm}_1]\!]^I \cup [\![\mathtt{sel\_stm}_2]\!]^I$*.*
- $[\![\mathtt{sel\_stm}_1 \text{ EXCEPT } \mathtt{sel\_stm}_2]\!]^I = [\![\mathtt{sel\_stm}_1]\!]^I \setminus [\![\mathtt{sel\_stm}_2]\!]^I$*.*

- $[\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k]\!]^I = \{(exp_1, \ldots, exp_k)\}$, *where* exp_i *denotes the mathematical evaluation of* $\mathtt{exp}_i$.

- $[\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m \text{ WHERE } \mathtt{cond}]\!]^I =$
  $\{(exp_1[\overline{a}/\overline{\mathtt{A}}], \ldots, exp_k[\overline{a}/\overline{\mathtt{A}}]) \quad | \quad \overline{a} \in I(\mathtt{R}_1) \times \ldots \times I(\mathtt{R}_m), cond[\overline{a}/\overline{\mathtt{A}}] \textit{ is satisfied}\}$,

  where $\overline{\mathtt{A}}$ *represents a sequence of attributes prefixed with their corresponding relation names, i.e., if* $\mathtt{A}^j_1, \ldots, \mathtt{A}^j_{r_j}$ *are the attributes of* $\mathtt{R}_j$, $1 \leq j \leq m$, *then* $\overline{\mathtt{A}}$ *is the complete sequence* $\mathtt{R}_1.\mathtt{A}^1_1, \ldots, \mathtt{R}_1.\mathtt{A}^1_{r_1}, \ldots, \mathtt{R}_m.\mathtt{A}^m_1, \ldots, \mathtt{R}_m.\mathtt{A}^m_{r_m}$; *the notation* $exp_j[\overline{a}/\overline{\mathtt{A}}]$, $1 \leq j \leq k$, *stands for the mathematical evaluation of* $\mathtt{exp}_j$, *after replacing the tuple* $\overline{a}$ *by* $\overline{\mathtt{A}}$; *and* $cond[\overline{a}/\overline{\mathtt{A}}]$ *denotes the evaluation of the Boolean expression* cond, *with the previous substitution.*

Next, for every i, an operator $T^{\mathtt{db}}_i$ over the set $\mathcal{I}^{\mathtt{db}}_i$ of interpretations of stratum i for db is defined. $T^{\mathtt{db}}_i$ is continuous, as stated in [2]. The least fixpoint of $T^{\mathtt{db}}_i$ is the interpretation giving meaning to the relations of db in the stratum i.

Definition 4. *The operator* $T^{\mathtt{db}}_i : \mathcal{I}^{\mathtt{db}}_i \longrightarrow \mathcal{I}^{\mathtt{db}}_i$ *transforms interpretations over* i *as follows. For every* $I \in \mathcal{I}^{\mathtt{db}}_i$ *and for every* $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$*:*

- $T^{\mathtt{db}}_i(I)(\mathtt{R}) = I(\mathtt{R})$, *if* $str(\mathtt{R}) < i$.

- $T^{\mathtt{db}}_i(I)(\mathtt{R}) = [\![\mathtt{sel\_stm}]\!]^I$, *if* $str(\mathtt{R}) = i$ *and* sel_stm *is the definition of* R *in* db.

- $T^{\mathtt{db}}_i(I)(\mathtt{R}) = \emptyset$, *if* $str(\mathtt{R}) > i$.

Proposition 1 (Continuity of $T^{\mathtt{db}}_i$)**.** Let $i \geq 1$ and $\{I_n\}_{n\geq 0}$ be a chain of interpretations in $\mathcal{I}^{\mathtt{db}}_i$ ($I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$). Then, $T^{\mathtt{db}}_i(\bigsqcup_{n\geq 0} I_n) = \bigsqcup_{n\geq 0} T^{\mathtt{db}}_i(I_n)$.

Therefore, the existence of a least fixpoint stratum by stratum is a direct consequence of the Knaster-Tarski fixpoint theorem [14].

Theorem 1. There is a fixpoint interpretation $fix^{\mathtt{db}} : \mathtt{RN}_{\mathtt{db}} \to \mathcal{P}(\mathcal{T})$, such that for every R $\in \mathtt{RN}_{\mathtt{db}}$, if sel_stm is the definition of R in db, then $fix^{\mathtt{db}}(\mathtt{R}) = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}}}$.

The interpretation $fix^{\mathtt{db}}$ defines the semantics of db. The construction of this fixpoint is stratum by stratum as follows:

The operator $T^{\mathtt{db}}_1$ has a least fixpoint, called $fix^{\mathtt{db}}_1$, which is $\bigsqcup_{n\geq 0} (T^{\mathtt{db}}_1)^n(\emptyset)$, the least upper bound of the sequence $\{(T^{\mathtt{db}}_1)^n(\emptyset)\}_{n\geq 0}$, where $(T^{\mathtt{db}}_1)^n(\emptyset)$ is the result of n successive applications of $T^{\mathtt{db}}_1$ to the empty interpretation.

Consider now the sequence $\{(T^{\mathtt{db}}_2)^n(fix^{\mathtt{db}}_1)\}_{n\geq 0}$ of interpretations in $(\mathcal{I}^{\mathtt{db}}_2, \sqsubseteq_2)$ greater than $fix^{\mathtt{db}}_1$. Using the definition of $T^{\mathtt{db}}_i$ and the fact that $fix^{\mathtt{db}}_1(\mathtt{R}) = \emptyset$ for every R such that $str(\mathtt{R}) \geq 2$, it is easy to prove (as for the stratum 1) that such sequence is a chain, $fix^{\mathtt{db}}_1 \sqsubseteq_2 T^{\mathtt{db}}_2(fix^{\mathtt{db}}_1) \sqsubseteq_2 T^{\mathtt{db}}_2(T^{\mathtt{db}}_2(fix^{\mathtt{db}}_1)) \sqsubseteq_2 \ldots, \sqsubseteq_2 (T^{\mathtt{db}}_2)^n(fix^{\mathtt{db}}_1), \ldots$ with least upper bound in $(\mathcal{I}^{\mathtt{db}}_2, \sqsubseteq_2)$, $\bigsqcup_{n\geq 0} (T^{\mathtt{db}}_2)^n(fix_1)$, that is the least fixpoint of $T^{\mathtt{db}}_2$ containing $fix^{\mathtt{db}}_1$, called $fix^{\mathtt{db}}_2$.

Now, if $k = max\{str(\mathtt{R}) \mid \mathtt{R} \in \mathtt{RN}_{\mathtt{db}}\}$, by proceeding successively, for every i, $1 < i \leq k$, a chain, $\{(T^{\mathtt{db}}_i)^n(fix^{\mathtt{db}}_{i-1})\}_{n\geq 0}$ can be defined, and a fixpoint of $T^{\mathtt{db}}_i$, $fix^{\mathtt{db}}_i = \bigsqcup_{n\geq 0} (T^{\mathtt{db}}_i)^n(fix^{\mathtt{db}}_{i-1})$, can be found. In addition, $fix^{\mathtt{db}}_1 \sqsubseteq_k \ldots \sqsubseteq_k fix^{\mathtt{db}}_k$. We call $fix^{\mathtt{db}}$ to $fix^{\mathtt{db}}_k$, since it contains the information of the whole database.

3 The Query Language of HR-SQL

As usual in SQL, users of an HR-SQL database can formulate queries by means of select statements. The novelty of the HR-SQL language w.r.t. R-SQL is the incorporation of hypothetical queries. The syntax of queries is defined as:

```
query    ::=  sel_stm | sel_hyp
sel_hyp  ::=  ASSUME hypo,...,hypo sel_stm
hypo     ::=  sel_stm [NOT] IN R
```

Example 2. *Consider the database of Example 1, and the query: how long does it take to arrive in Valverde from Madrid, if boat links that take more than one hour are not considered? It can be expressed in HR-SQL as:*

```
ASSUME SELECT * FROM boat WHERE boat.time > 1 NOT IN link
SELECT travel.time FROM travel
     WHERE travel.ori = 'MAD' and travel.des = 'VDE'
```

From the logical point of view, a hypothetical query can be interpreted as an intuitionistic implication: it represents the value of the consequent assuming the antecedent. Next we formalize this idea.

3.1 The Semantics of a Query

As usual, the answer of a query is identified with the set of tuples that satisfy such a query. So, for a stratifiable database definition db, this answer corresponds to the interpretation of the query w.r.t. the fixpoint of db. The following definition formalizes this concept for the different cases of queries. In the case of a hypothetical query, to reflect the changes introduced in the current database assuming the hypothesis, we will use the notation db[R sch := sel_stm′/R sch := sel_stm] to denote the database definition that results from the database db by replacing the relation definition R sch := sel_stm by R sch := sel_stm′. In addition, *sel*(query) denotes the select statement of query. More precisely *sel*(sel_stm) = sel_stm and *sel*(ASSUME $\mathtt{hypo}_1, \ldots, \mathtt{hypo}_k$ sel_stm) = sel_stm.

For readability, we give the definition only for the case of one assumption; for a sequence of assumptions it is obtained as a simple sequential extension, considering a sequence of such replacements, as shown in Example 3 later.

Definition 5. *Let* query *be a query for* db*. Its* answer w.r.t. db*, denoted by* $[\![\mathtt{query}]\!]_{\mathtt{db}}$*, is defined by cases:*

Simple query: $[\![\mathtt{sel\_stm}]\!]_{\mathtt{db}} = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}}}$.

Hypothetical query: If R sch := $\mathtt{sel\_stm}_{\mathtt{R}}$ *is the definition of* R *in* db*, then:*

- $[\![$ASSUME sel_stm′ IN R sel_stm$]\!]_{\mathtt{db}} = [\![\mathtt{sel_stm}]\!]^{fix^{\mathtt{db}'}}$, *where* $\mathtt{db}' = \mathtt{db}[$R sch := $\mathtt{sel\_stm}_{\mathtt{R}}$ UNION sel_stm′/ R sch := $\mathtt{sel\_stm}_{\mathtt{R}}]$.
- $[\![$ASSUME sel_stm′ NOT IN R sel_stm$]\!]_{\mathtt{db}} = [\![\mathtt{sel_stm}]\!]^{fix^{\mathtt{db}'}}$, *where* $\mathtt{db}' = \mathtt{db}[$R sch := $\mathtt{sel\_stm}_{\mathtt{R}}$ EXCEPT sel_stm′/ R sch := $\mathtt{sel\_stm}_{\mathtt{R}}]$.

Example 3. *Let* db *be the following database definition (for simplicity, we omit the schema* A int *for all the relations):*

```
R1:= SELECT 1 UNION SELECT 2 UNION SELECT 3;
R2:= sel_stm_R2
```

where $\texttt{sel\_stm}_{\texttt{R2}} \equiv$ SELECT 1 UNION SELECT 3 UNION SELECT 5
EXCEPT SELECT R1.A FROM R1 WHERE R1.A=1 OR R1.A=2;
R3:= SELECT R2.A FROM R2 UNION SELECT R3.A*2 FROM R3 WHERE R3.A<5;

Consider the following hypothetical query:

query ≡ ASSUME SELECT R1.A FROM R1 WHERE R1.A < 3 IN R2,
SELECT 3 NOT IN R2
SELECT R3.A FROM R3

Then $[\![\texttt{query}]\!]_{\texttt{db}} = [\![\text{SELECT R3.A FROM R3}]\!]^{fix^{\texttt{db}'}}$, *where* $\texttt{db}' = (\texttt{db})\theta\sigma$ *being:*
$\theta = [\texttt{R2 := sel\_stm}'_{\texttt{R2}}/\ \texttt{R2 := sel\_stm}_{\texttt{R2}}]$,
$\sigma = [\texttt{R2 := sel\_stm}'_{\texttt{R2}}$ EXCEPT SELECT 3$/\texttt{R2 := sel_stm}'_{\texttt{R2}}]$,
$\texttt{sel\_stm}'_{\texttt{R2}} \equiv \texttt{sel\_stm}_{\texttt{R2}}$ UNION SELECT R1.A FROM R1 WHERE R1.A < 3.

Therefore db′ *is the following database:*

R1:= SELECT 1 UNION SELECT 2 UNION SELECT 3;
R2:= ((SELECT 1 UNION SELECT 3 UNION SELECT 5
EXCEPT SELECT R1.A FROM R1 WHERE R1.A=1 OR R1.A=2)
UNION SELECT R1.A FROM R1 WHERE R1.A<3) EXCEPT SELECT 3;
R3:= SELECT R2.A FROM R2 UNION SELECT R3.A*2 FROM R3 WHERE R3.A<5;

The computation of a simple query for an existing database is easy, because the value of $[\![\texttt{sel\_stm}]\!]_{\texttt{db}}$ is $[\![\texttt{sel\_stm}]\!]^{fix^{\texttt{db}}}$, and $fix^{\texttt{db}}$ is known and coincides with the instance of the database. The case of a hypothetical query sel_hyp requires additional explanation, its meaning is the interpretation of a select statement w.r.t. a new database db′, where some relations have changed because the assumptions are incorporated to the corresponding relations. db′ must be a stratifiable database in order to define the interpretation $fix^{\texttt{db}'}$. By taking advantage of the stratified semantics, the computation of $fix^{\texttt{db}'}$ can be simplified:

First, the dependency graph $DG_{\texttt{db}'}$ is an extension of $DG_{\texttt{db}}$, because $\texttt{RN}_{\texttt{db}'} = \texttt{RN}_{\texttt{db}}$, and every relation definition of db′ is in db, but the new relation definition $\texttt{R sch := sel\_stm}_{\texttt{R}}$ UNION|EXCEPT $\texttt{sel\_stm}'$. The edges from the relations inside $\texttt{sel\_stm}_{\texttt{R}}$ to R are already in $DG_{\texttt{db}}$. So $DG_{\texttt{db}'}$ can be built from $DG_{\texttt{db}}$ as follows: For every $\texttt{R}' \in \texttt{RN}_{\texttt{sel\_stm}'}$, an edge from R′ to R is added; it is negatively labeled in the EXCEPT case or if $\texttt{R}' \in \texttt{RN}^{\neg}_{\texttt{sel\_stm}'}$. A stratification for db′, $str' : \texttt{RN}_{\texttt{db}'} \to \{1, \ldots, n\}$, if it exists, satisfies $str'(\texttt{R}) \geq str(\texttt{R})$, since (as we have remarked already) the select statement that defines R in db′, contains the select statement $\texttt{sel\_stm}_{\texttt{R}}$, which defines R in db.

Second, in order to obtain $[\![sel(\texttt{sel\_hyp})]\!]^{fix^{\texttt{db}'}}$, it is only necessary to compute $fix^{\texttt{db}'}(\texttt{R}')$ for the relations R′ such that the relations in $\texttt{RN}_{sel(\texttt{sel\_hyp})}$ depend on R′. In addition, $fix^{\texttt{db}'}$ has not to be computed from stratum 1, as we will see. Let $i = str'(\texttt{R})$ ($i = min\{str'(\texttt{R}_j) | 1 \leq j \leq k\}$ in the general case, if assumptions for the relations $\texttt{R}_1, \ldots, \texttt{R}_k$ are considered), then $fix^{\texttt{db}'}(\texttt{R}') = fix^{\texttt{db}}(\texttt{R}')$, for every R′ with $str'(\texttt{R}') < i$. And let $S = \{\texttt{R}'' \mid \texttt{R}' \in \texttt{RN}_{sel(\texttt{sel\_hyp})}$ and R′ depends on $\texttt{R}''\}$, then $fix^{\texttt{db}'}$ can be obtained from $fix^{\texttt{db}}$ in the following way:

1. Compute $fix_i^{\texttt{db}'}(\texttt{R}')$ from $fix_{i-1}^{\texttt{db}}$ for every relations $\texttt{R}' \in S$ and $str'(\texttt{R}') = i$.
2. Compute $fix_j^{\texttt{db}'}(\texttt{R}')$ from $fix_{j-1}^{\texttt{db}'}$ for the relations $\texttt{R}' \in S$ and $str'(\texttt{R}') = j$, $j = i + 1$ .. $str'(sel(\texttt{sel\_hyp}))$.

Example 4. *Consider the stratifiable database* db *and the query of Example 3. Let str be a stratification for* db*, such that* $str(\texttt{R1}) = 1$, $str(\texttt{R2}) = 2$, $str(\texttt{R3}) = 3$. *In this case, str is also a stratification for the modified database* db′*, detailed in Example 3, needed to answer to* query.

It is easy to check that:
$fix^{\mathtt{db}}(\mathtt{R1}) = \{(1),(2),(3)\}$, $fix^{\mathtt{db}}(\mathtt{R2}) = \{(3),(5)\}$, $fix^{\mathtt{db}}(\mathtt{R3}) = \{(3),(5),(6)\}$.
In order to obtain $fix^{\mathtt{db}'}$*, notice that* $\mathtt{RN}_{sel(\mathtt{query})} = \{\mathtt{R3}\}$*. So* $S = \{\mathtt{R}'' \mid \mathtt{R}' \in \{\mathtt{R3}\}$ *and* $\mathtt{R}'$ *depends on* $\mathtt{R}''\} = \{\mathtt{R1}, \mathtt{R2}, \mathtt{R3}\}$*, but the computation can start at stratum 2 =* $str(\mathtt{R2})$*, with* $fix_1^{\mathtt{db}'} = fix_1^{\mathtt{db}}$*.* R2 *is the only relation in S in stratum 2.*
$fix_2^{\mathtt{db}'}(\mathtt{R2}) = \{(1),(2),(5)\}$.
Similarly, for stratum 3, only $fix_3^{\mathtt{db}'}(\mathtt{R3})$ *must be computed to get the answer, even in the case that* db *had other relations in this stratum.*
$fix_3^{\mathtt{db}'}(\mathtt{R3}) = \{(1),(2),(4),(5),(8)\} = [\![\text{SELECT R3.A FROM R3}]\!]^{fix^{\mathtt{db}'}} = [\![\mathtt{query}]\!]_{\mathtt{db}}$.

4 The View Definition Language of HR-SQL

In this section we extend the definition language by allowing the definition of views, which essentially consists of assigning names to queries in order to use them as relation names inside other queries, or inside itself to express recursive queries. The syntax is as follows:

```
vd    ::=  view ...  view
view  ::=  V sch := sel_stm; | HV sch := sel_hyp;
```

We use V for names of views that are defined by a non hypothetical query, and HV for hypothetical views. From now on, those symbols can be considered as elements of the set $\mathtt{RN}_{\mathtt{db}}$ as relation names.

We say that vd is *a definition of views for* db if the involved names in it are relation names of db or view names defined in vd. Mutual recursive definitions are allowed among non hypothetical views. Then their names can occur inside the definition of any view (hypothetical or not). Every hypothetical view can be recursive but its name cannot appear inside the definition of other views, which means that in a definition of views of the form:

$\mathtt{V}_1\ \mathtt{sch}_1\ \mathtt{:=}\ \mathtt{sel\_stm}_1\mathtt{;}\ \ldots\ \ \mathtt{V}_m\ \mathtt{sch}_m\ \mathtt{:=}\ \mathtt{sel\_stm}_m\mathtt{;}$
$\mathtt{HV}_1\ \mathtt{sch}_1\ \mathtt{:=}\ \mathtt{sel\_hyp}_1\mathtt{;}\ \ldots\ \ \mathtt{HV}_r\ \mathtt{sch}_r\ \mathtt{:=}\ \mathtt{sel\_hyp}_r\mathtt{;}$

for every $j = 1..m$, $\mathtt{V}_j$ can occur everywhere; for every $j = 1..r$, $\mathtt{HV}_j$ can occur inside $sel(\mathtt{sel\_hyp}_j)$, but not in $\mathtt{sel\_stm}_1, \ldots, \mathtt{sel\_stm}_m$, $\mathtt{sel\_hyp}_k$, if $k \neq j$, nor in the assumption part of $\mathtt{sel\_hyp}_j$.

Example 5. *Referring to Example 1, assume there is a volcanic eruption in El Hierro and the airspace must be closed in the archipelago, as well as the bus in this island. But a boat from El Hierro to La Palma is added. The hypothetical view below can be defined in HR-SQL to represent the reachable cities in the Canary islands in this situation.*

```
reachable(ori varchar(10),des varchar(10)) := ASSUME
    (SELECT * FROM bus WHERE bus.ori = 'VDE'
     UNION SELECT * FROM flight) NOT IN link,
     SELECT 'RES','SPC',1.5 IN boat
  SELECT link.ori, link.des FROM link UNION
  SELECT link.ori, reachable.des FROM link, reachable
   WHERE link.des = reachable.ori;
```

4.1 The Semantics of a Definition of Views

A view name identifies a query, so the meaning of a definition of views vd sets the correspondence between every view name in vd and the interpretation of the corresponding query. But this

interpretation must consider the original database definition extended with the views defined in vd as new relations. As we will show, stratification must be extended to assign a stratum to every view name. Next, these ideas are formalized. First we consider the definition of a simple view, then we will generalize it to the definition of a sequence of views.

Definition 6. *Let* V sch := sel_stm *be the definition of a non hypothetical view for* db. *The* meaning of V w.r.t. db*, denoted by* $[\![\mathtt{V}]\!]_{\mathtt{db}}$*, is equal to* $[\![\mathtt{sel\_stm}]\!]_{\mathtt{db}'}$*, where* db′ *is the result of extending* db *with* V sch := sel_stm *as a new relation.*

Let HV sch := sel_hyp *be the definition of a hypothetical view for* db. *The* meaning of HV w.r.t. db*, denoted by* $[\![\mathtt{HV}]\!]_{\mathtt{db}}$*, is equal to* $[\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}'}$*, where* db′ *is the result of extending* db *with* HV sch := *sel*(sel_hyp) *as a new relation.*

In V sch := sel_stm, the value $[\![\mathtt{V}]\!]_{\mathtt{db}} = [\![\mathtt{sel\_stm}]\!]_{\mathtt{db}'} = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}'}}$ depends on the fixpoint of a new database definition which should be stratifiable. db′ is equal to db extended with V sch := sel_stm, it will be non stratifiable if V appears in an EXCEPT clause inside sel_stm, but in the other case, the fixpoint of the new database will be equal to the one of db, except for the new relation V. Notice that $\mathtt{RN}_{\mathtt{db}'} = \mathtt{RN}_{\mathtt{db}} \cup \{\mathtt{V}\}$, and no relation defined in db may depend on V. So, if k is the maximum stratum of db (with n relations), then a stratification str' for db′ can be defined as $str' : \mathtt{RN}_{\mathtt{db}'} \to \{1, \ldots, n+1\}$, with $str'(\mathtt{R}) = str(\mathtt{R})$ for every $\mathtt{R} \in \mathtt{RN}_{\mathtt{db}}$, and $str'(\mathtt{V}) = k+1$. Hence, for $i = 1..k$, $fix_i^{\mathtt{db}'} = fix_i^{\mathtt{db}}$. Therefore: $fix^{\mathtt{db}'} = fix_{k+1}^{\mathtt{db}'} = \bigsqcup_{m \geq 0}(T_{k+1}^{\mathtt{db}'})^m(fix^{\mathtt{db}})$, so $fix^{\mathtt{db}'}$ is an extension of the known $fix^{\mathtt{db}}$, and only the last stratum $k+1$ for the relation V (the only one in this stratum) must be calculated to find $[\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}'}} = fix_{k+1}^{\mathtt{db}'}(\mathtt{V})$.

The semantics for the case HV sch := ASSUME sel_stm′ [NOT] IN R sel_stm requires to modify the original database in two ways:

1. $[\![\mathtt{HV}]\!]_{\mathtt{db}} = [\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}'}$, according to Definition 6, where db′ results from extending db with HV sch := sel_stm.
2. $[\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}'} = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}''}}$, in accordance with Definition 5, where $\mathtt{db}'' = \mathtt{db}'[\mathtt{R\ sch := sel\_stm}_{\mathtt{R}}$ UNION | EXCEPT $\mathtt{sel\_stm}'/\mathtt{R\ sch := sel\_stm}_{\mathtt{R}}]$.

In 1, the original database is extended with a new relation, HV. Notice that, considering HV as a relation identifier, the added definition, HV sch := sel_stm, is syntactically correct (however HV sch := sel_hyp is not allowed as a relation definition). In 2, the assumption is incorporated to the corresponding relation, as explained in Section 3.1. So, the new relation definitions in db″ are:

HV sch := *sel*(sel_hyp); R sch := $\mathtt{sel\_stm}_{\mathtt{R}}$ UNION | EXCEPT sel_stm′;

Then, the dependency graph $DG_{\mathtt{db}''}$ can be built from $DG_{\mathtt{db}}$ adding new edges to the relation R, as explained before. But there is also a new node HV an new edges: For every $\mathtt{R}' \in \mathtt{RN}_{sel(\mathtt{sel\_hyp})}$, an edge from R′ to HV, that is negatively labelled if $\mathtt{R}' \in \mathtt{RN}^{\neg}_{sel(\mathtt{sel\_hyp})}$.

As for the non hypothetical case, a stratification of db″, $str' : \mathtt{RN}_{\mathtt{db}''} \to \{1, \ldots, n+1\}$, if it exists, may assign the stratum $k+1$ to HV. $fix_k^{\mathtt{db}''}$ can be computed as explained in Section 3.1 for hypothetical queries, and the computation of $fix_{k+1}^{\mathtt{db}''}$ will consider only HV, and $fix_{k+1}^{\mathtt{db}''}(\mathtt{HV}) = [\![\mathtt{sel\_stm}]\!]^{fix^{\mathtt{db}''}}$.

Example 6. *Consider the database* db *of Example 3 and the hypothetical view:*

```
HV A int := ASSUME
     SELECT R1.A FROM R1 WHERE R1.A < 3 IN R2,  SELECT 3 NOT IN R2
 SELECT R3.A FROM R3 UNION SELECT HV.A*3 FROM HV WHERE HV.A < 3;
```

Following Definition 6, $[\![\texttt{HV}]\!]_{\texttt{db}} = [\![\text{SELECT R3.A FROM R3 UNION SELECT HV.A*3 FROM HV WHERE HV.A<3}]\!]^{fix^{\texttt{db}''}}$*, where* db″ *is an extension of the database* db′ *of Example 3 with:*

```
HV A int := SELECT R3.A FROM R3 UNION
            SELECT HV.A*3 FROM HV WHERE HV.A < 3;
```

A function str' *extending* str *in such a way that* $str'(\texttt{HV}) = 4$ *is a stratification of the new database. For* $1 \leq i \leq 3$, $fix_i^{\texttt{db}''} = fix_i^{\texttt{db}'}$*, which appear in Example 4. Then, since* $[\![\texttt{HV}]\!]_{\texttt{db}}$ *coincides with* $fix^{\texttt{db}''}(\texttt{HV})$*, it is only necessary to calculate* $fix_4^{\texttt{db}''}(\texttt{HV}) = (\bigsqcup_{m\geq 0}(T_4^{\texttt{db}''})^m(fix_3^{\texttt{db}''}))(\texttt{HV}) = \{(1),(2),(3),(4),\ (5),(6),(8)\}$.

Next we deal with the case of simultaneous view definitions for a database db. The idea is that the semantics of vd associates to every view name in vd, the interpretation of the query that defines the view. But, if there is more than one non hypothetical view definition in vd, it is not valid to identify $[\![\texttt{V}]\!]_{\texttt{db}}$ with $[\![\texttt{sel\_stm}]\!]_{\texttt{db}'}$, being db′ the result of extending db with `V sch := sel_stm`. This is because other names defined in vd distinct of V can occur inside sel_stm, while they are not defined in db′. Then the semantics of vd is defined as follows:

Definition 7. *Let* db *be a database and let*

```
vd ::= V1 sch1 := sel_stm1; ... ; Vm schm := sel_stmm;
       HV1 sch1 := sel_hyp1; ... ; HVr schr := sel_hypr;
```

be a definition of views for db*. The semantics of* vd *is defined as the mapping that associates* $\texttt{V}_j$ *to* $[\![\texttt{V}_j]\!]_{\texttt{db}'}$*, for* $j = 1..m$*, and* $\texttt{HV}_j$ *to* $[\![\texttt{HV}_j]\!]_{\texttt{db}'}$*, for* $j = 1..r$*, where* db′ *is the result of extending* db *with:*

```
V1 sch1 := sel_stm1; ... ; Vm schm := sel_stmm;
```

Notice that, according to Definition 6, $[\![\texttt{V}_j]\!]_{\texttt{db}'} = [\![\texttt{sel\_stm}_j]\!]_{\texttt{db}''}$ for every $j = 1..m$, where db″ is the result of extending db′ with $\texttt{V}_j\texttt{sch}_j$:= $\texttt{sel\_stm}_j$, but this definition is already in db′, so db″ = db′. Since, $\texttt{HV}_1, \ldots, \texttt{HV}_r$ do not appear in $\texttt{sel\_stm}_j$, their definitions are not required in db′. But for every $1 \leq j \leq r$, $[\![\texttt{HV}_j]\!]_{\texttt{db}'} = [\![\texttt{sel\_hyp}_j]\!]_{\texttt{db}''}$, where db″ is the result of extending db′ with $\texttt{HV}_j$ $\texttt{sch}_j$:= $sel(\texttt{sel\_hyp}_j)$, allowing $\texttt{HV}_j$ to be recursive.

In order to compute the answer of every view included in a simultaneous definition, hypothetical views can be relegated to process the others. As in the simple case, db′ must be stratifiable. In the practice, if db′ is stratifiable, a stratification str' for db′, such that $str'(\texttt{V}_j) > n$ for every $1 \leq j \leq m$ can be found. Then the interpretation $fix^{\texttt{db}'}$ can be obtained stratum by stratum, starting from $fix^{\texttt{db}}$, as in the simple case. Now, every hypothetical view can be treated separately, starting each time with $fix^{\texttt{db}'}$ as initial interpretation, and processing each view as in the simple case.

5 The HR-SQL System

We present a SWI-Prolog implementation for the HR-SQL language adapted for IBM DB2. The system, with a bundle of examples, is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/HR-SQL`. The structure of the system is depicted in Figure 5. The interface consists of a prompt '`hr-db2 =>`' which works as an extension of the command interpreter of DB2. The user can submit any DB2 input to manage an existing database (label A in Figure 5), and also the following ones provided by HR-SQL (label B in Figure 5):

- `load_db` <*db_file*> loads an HR-SQL database definition from a file and computes the corresponding fixpoint. The resulting tuples for the relations are stored as DB2 tables.
- `load_vd` <*vd_file*> loads an HR-SQL definition of views from a file, computes the values for

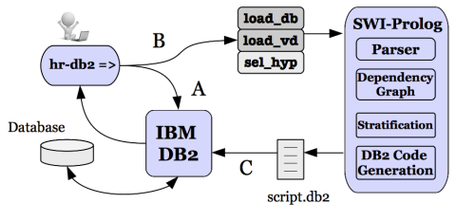


Figure 2: The HR-SQL System.

each view, and materializes them as DB2 tables.

• A hypothetical query written in HR-SQL syntax (`sel_hyp`), which is submitted to the system and recognized as such because it starts with ASSUME.

These new statements are preprocessed by the SWI-Prolog module as shown in Figure 5. After parsing, the dependency graph is built, a stratification is generated (if it exists; an error is thrown otherwise). The current algorithm to compute the stratification tries to minimize the number of relations in each strata. This allows to improve the efficiency of the fixpoint computation w.r.t. [2], because now each stratum i contains only those mutually recursive relations, avoiding to process the rest of them in each iteration of the fixpoint operator at stratum i. After the stratification, an SQL PL script is produced as will be explained in Section 5.1. This output is executed by DB2 (label C in Figure 5). The code generation for hypothetical views needs an additional process which is shown in Section 5.2.

5.1 Computing the Fixpoint

Figure 3 shows the algorithm for generating the DB2 database corresponding to the fixpoint of an HR-SQL database definition. It produces the SQL statements (CREATE and INSERT) needed to build such a database. This version enhances the one in [2] with the functions *in* and *out* which will be explained later.

```
1  for all R ∈ RN_db do CREATE TABLE R sch;
2  i := 1
3  while i ≤ numStr do
4    for all R ∈ RN_i do INSERT INTO R out(sel_stm_R);
5    repeat
6      size := rel_size(RN_i)
7      for all R ∈ RN_i do
8        INSERT INTO R in(sel_stm_R) EXCEPT SELECT * FROM R;
9    until size = rel_size(RN_i)
10   i := i + 1
```

Figure 3: Algorithm to Compute the Fixpoint

The algorithm considers a concrete stratification for the database where *numStr* denotes the number of strata and $\mathtt{NR}_i$ the set of relations of stratum i. First of all, a table is created for each relation $\mathtt{R\ sch} := \mathtt{sel\_stm_R}$ of the database (line 1). Then, the external *while* (lines 3-10) computes successively the fixpoints $fix_1^{\mathtt{db}}, fix_2^{\mathtt{db}}, \ldots, fix_{numStr}^{\mathtt{db}}$. According to the theory, each

$fix_i^{\mathtt{db}}$ is calculated for every relation of $\mathtt{NR}_i$, by iterating the operators $T_i^{\mathtt{db}}$ of Definition 3, i.e., the *repeat* (lines 5-9) at iteration n computes $(T_i^{\mathtt{db}})^n(fix_{i-1})$. The loop is iterated while some tuple is added to the tables of the current stratum; the variable *size* is used to check if some tuple is added to some relation of the current stratum.

This algorithm improves the efficiency of the introduced in [2] by reducing the work in the iterations of the *repeat* with the functions *in* and *out*. The idea is that the iteration of the operator $T_i^{\mathtt{db}}$ is only needed for recursive relations; in fact, only for the recursive fragment of the select statements defining those relations. The functions *in* and *out* split each `sel_stm` into the (recursive) fragment that must be used in the INSERT statements inside the loop (line 8), and the fragment that can be processed before the loop, as the base case of the recursive definition (line 4). The *in* and *out* fragments of a `sel_stm` can be easily determined using the stratum of its components because, as mentioned before, the stratification is such that if a relation `R` in stratum i depends on another relation $\mathtt{R}'$, then the stratum of $\mathtt{R}'$ is lower than i, so it must be previously computed, or it is exactly i (if they are mutually recursive) and both relations must be computed simultaneously. Therefore, if for instance `R := sel_stm`$_1$ UNION `sel_stm`$_2$, $str(\mathtt{R}) = i$, and $str(\mathtt{sel\_stm}_1) < i$, then `sel_stm`$_1$ will be part of the *out* fragment, and the corresponding tuples can be inserted before the loop, because the involved relations are already computed in the computation of a previous stratum. Functions *in* and *out* are defined by recursion on the structure of `sel_stm`. For example, if `sel_stm` $\equiv$ `ss`$_1$ EXCEPT `ss`$_2$, and $str(\mathtt{sel\_stm}) = i$, then $str(\mathtt{ss}_2) < i$, so:

$in(\mathtt{sel\_stm}) = in(\mathtt{ss}_1)$ EXCEPT $\mathtt{ss}_2$; $\quad out(\mathtt{sel\_stm}) = out(\mathtt{ss}_1)$ EXCEPT $\mathtt{ss}_2$.

5.2 Computing Hypothetical Views

The SQL PL script generated to process views follows the ideas of Section 4.1. We use the view `reachable` of Example 5 to illustrate the system steps to solve a hypothetical view definition. It is interesting as it is a recursive definition containing positive and negative assumptions.

First of all, the system extends the original dependency graph with the new edges due to hypothetical assumptions: two negatively labeled edges to `link`, one from `bus`, and another from `flight`. Due to the expanded form of stratification we have defined, the stratification for the original database is also a stratification for the new one. Following the explanations of Section 3.1, the system looks for those relations that must be recomputed to obtain the tuples of the view `reachable`, in this case only `boat` and `link`. The algorithm that generates the SQL statements, for computing these relations and the new view, is quite similar to that presented in Figure 3 to compute the fixpoint of a database. Next we explain the differences following the example.

The relations needed to compute the view are locally created and recomputed using temporary tables, and the computation will start at stratum $i = min\{str(\mathtt{boat}), str(\mathtt{link})\}$:

```
DECLARE GLOBAL TEMPORARY TABLE link AS link;
DECLARE GLOBAL TEMPORARY TABLE boat LIKE boat;

INSERT INTO SESSION.boat
 ((SELECT 'TFS','GMZ',1) UNION (SELECT 'GMZ','VDE',1.5) UNION
  (SELECT 'SPC','TFN',2 l) UNION (SELECT 'RES','SPC',1.5));

INSERT INTO SESSION.link
 (SELECT * FROM flight UNION SELECT * FROM SESSION.boat UNION
  SELECT * FROM bus EXCEPT (SELECT * FROM bus WHERE bus.ori = 'VDE')
  UNION SELECT * FROM flight);
```

Temporary tables are prefixed with `SESSION`. For processing a hypothetical view `HV := sel_hyp`$_{\mathtt{HV}}$, the script to compute the tuples of `HV` will consider the definition `HV := sel_stm`, where `sel_stm`

results from replacing `R` by `SESSION.R` in $sel(\texttt{sel\_hyp}_{\texttt{HV}})$. The tuples for `reachable` are materialized and stored, then the temporary tables are discarded. Temporary tables are adequate as they are in-memory data structures.

The computation of hypothetical queries follows the same steps, but instead of creating tables, a cursor is used to obtain the answer without materializing it.

6 Conclusions and Future Work

We have designed a practical, formally-supported SQL system, porting some techniques from the deductive database field to the relational one. Thus, we provide an original way to give semantics to SQL languages supporting recursion. In addition our system allows both less-limited recursion (w.r.t. current SQL systems) and hypothetical reasoning (as a novel addition to such systems), acting as a front-end to DB2. Although targeted to this system, our work can be straightforwardly applied to any other SQL system. However, it can be improved in a number of ways: With respect to recursion, in-memory indexing can be applied for small search keys. These keys can be identified as the candidate keys derived from explicit functional dependencies (as already allowed in DB2) and primary keys. Also, both general and particular optimization methods can be applied to our work. For the first, the differential semi-naïve algorithm [15] allows to save tuples in recursive joins along fixpoint iterations. For the second sort of method, already-known linear recursion optimizations [11] can also be applied by analyzing the dependency graph and easily detecting such cases. With respect to hypothetical queries and views, we plan to extend the definition language, allowing mutual recursion in hypothetical views. Finally, we can extend this work by allowing not only materialized views, but also regular views. For this, table functions (cf. IBM DB2 concepts) can be used as a natural construction to build HR-SQL query results on-the-fly.

References

[1] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández. An extended constraint deductive database: Theory and implementation. *The Journal of Logic and Algebraic Programming*, 2013.

[2] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández. Formalizing a Broader Recursion Coverage in SQL. In *Symposium on Practical Aspects of Declarative Languages (PADL'13)*, volume 7752 of *LNCS*, 2013. In Press.

[3] A. J. Bonner. Hypothetical datalog: Negation and linear recursion. In *The ACM Symposium on the Principles of Database Systems (PODS)*, pages 286–300, 1989.

[4] A. J. Bonner and M. Kifer. Transaction logic programming. In *ICLP*, pages 257–279, 1993.

[5] A. J. Bonner and L. T. McCarty. Adding negation-as-failure to intuitionistic logic programming. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming, Proc. of the North American Conference*, pages 681–703. The MIT Press, 1989.

[6] H. Christiansen and T. Andreasen. A Practical Approach to Hypothetical Database Queries. In *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 340–355. Springer, 1998.

[7] E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):377–390, June 1970.

[8] C. J. Date. *SQL and relational theory: how to write accurate SQL code.* O'Reilly, Sebastopol, CA, 2009.

[9] T. Griffin and R. Hull. A framework for implementing hypothetical queries. In *SIGMOD Conference*, pages 231–242, 1997.

[10] W. H. Inmon. *Building the data warehouse*. QED Information Sciences, Inc., Wellesley, MA, USA, 2005.

[11] C. Ordonez. Optimization of Linear Recursive Queries in SQL. *IEEE Transactions on Knowledge and Data Engineering*, 22(2):264–277, 2010.

[12] F. Sáenz-Pérez. Datalog Educational System, October 2011. `http://des.sourceforge.net/`.

[13] M. Stonebraker and K. Keller. Embedding expert knowledge and hypothetical data bases into a data base system. In *The 1980 ACM SIGMOD International Conference on Management of Data*, SIGMOD '80, pages 58–66. ACM, 1980.

[14] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[15] J. Ullman. *Principles of Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1989.

Electronic Communications of the EASST
Volume 64 (2013)

Proceedings of the XIII Spanish Conference on Programming and Computer Languages (PROLE 2013)

R-SQL: An SQL Database System with Extended Recursion[1]

Gabriel Aranda, Susana Nieva, Fernando Sáenz-Pérez and
Jaime Sánchez-Hernández

18 pages

Guest Editors: Clara Benac Earle, Laura Castro, Lars-Åke Fredlund
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

[1] This work has been partially supported by the Spanish projects TIN2013-44742-C4-3-R (CAVI-ART), TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

ECEASST

R-SQL: An SQL Database System with Extended Recursion[†]

Gabriel Aranda[1], Susana Nieva[1], Fernando Sáenz-Pérez[2] and Jaime Sánchez-Hernández[1]

[1] Dept. Sistemas Informáticos y Computación, UCM, Spain
[2] Dept. Ingeniería del Software e Inteligencia Artificial, UCM, Spain
garanda@fdi.ucm.es, nieva@sip.ucm.es, fernan@sip.ucm.es, jaime@sip.ucm.es

Abstract:

The relational database language SQL:1999 standard supports recursion, but this approach is limited to the linear case. Moreover, mutual recursion is not supported, and negation cannot be combined with recursion. We designed the language R-SQL to overcome these limitations in [ANSS13], improving termination properties in recursive definitions. In addition we developed a proof of concept implementation of an R-SQL system. In this paper we describe in detail an improved system enhancing performance. It can be integrated into existing RDBMS's, extending them with the aforementioned benefits of R-SQL. The system processes an R-SQL database definition obtaining its extension in tables of an RDBMS (such as PostgreSQL and DB2). It is implemented in SWI-Prolog and it produces a Python script that, upon execution, computes the result of the R-SQL relations. We provide some performance results showing the efficiency gains w.r.t. the previous version. We also include a comparative analysis including some representative relational a deductive systems.

Keywords: Databases, SQL, Recursion, Fixpoint Semantics

1 Introduction

Recursion is a powerful tool nowadays included in almost all programming systems. However, for current implementations of the declarative programming language SQL, this tool is heavily compromised or even not supported at all (MySQL, MS Access, ...) Those systems including recursion suffer from several drawbacks. Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Mutual recursion, and query solving involving an EXCEPT clause are not supported. In general, termination is manually controlled by limiting the number of iterations instead of detecting that there are no further opportunities to develop new tuples. Duplicate discarding is not supported and, so, queries that are actually terminating are not detected as such.

Starburst [MP94] was the first non-commercial RDBMS to implement recursion whereas IBM DB2 was the first commercial one. ANSI/ISO Standard SQL:1999 included for the first time

[†] This work has been partially supported by the Spanish projects TIN2013-44742-C4-3-R (CAVI-ART), TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

recursion in SQL. Today, we can find recursion in several systems: IBM DB2, Oracle, MS SQL Server, HyperSQL and others with the aforementioned limitations.

In [ANSS13] we proposed a new approach, called R-SQL, aimed to overcome these limitations and others, allowing in particular cycles in recursive definitions of graphs and mutually recursive relation definitions. In order to combine recursion and negation, we applied ideas from the deductive database field, such as stratified negation, based on the definition of a dependency graph between the relations involved in the database [Ull89]. We developed a formal framework following the original relational data model [Cod70], therefore avoiding both duplicates and nulls (as encouraged by [Dat09]). We used a stratified fixpoint semantics and we presented an R-SQL database system as a prototype implementing such formal framework. The system can be downloaded from `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQL`. In this work, we describe in detail an improved version enhancing performance. The system processes an R-SQL database definition obtaining its extension in tables of an RDBMS (such as PostgreSQL and DB2). It is implemented in SWI-Prolog and it generates a Python script that, upon execution, computes the result of the R-SQL relations. The improvements in efficiency relies on a new stratification and a more elaborated version of the fixpoint calculation algorithm that allows to avoid recomputations along iterations. The new system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQLplus`. In addition, we experiment with some previously proposed optimizations [Ull89] to improve the performance of the fixpoint computation.

Related academic approaches include DLV^{DB} [TLLP08], LDL++ [AOT+03] (now abandoned and replaced by DeALS, which does not refer to SQL queries up to now), and DES [SP13]. The first one, resulting of a spin-off at Calabria University, is the closer to our work as it produces SQL code to be executed in the external database with a semi-naïve strategy, but lacks formal support for its proposal, and it does not describe non-linear recursion. Last two ones also allow connecting to external databases, but processing of recursive SQL queries are in-memory.

The paper is organized as follows: In Section 2 we recall the syntax and the meaning of R-SQL database definitions. Section 3 describes the system, including the new form of stratification, the fixpoint algorithm, and some performance measurements, showing the efficiency gains for several optimizations. We also include a comparative analysis including some representative relational and deductive systems. Conclusions and future work are summarized in Section 4.

2 Introducing R-SQL

In this section, we present an overview of the language R-SQL, which is focused on the incorporation of recursive relation definitions. The idea is simple and effective: A relation is defined with an assignment operation as a named query (view) that can contain a self reference, i.e., a relation `R` can be defined as `R sch` := `SELECT...FROM ... R ...`, where `sch` is the relation schema.

2.1 The Definition Language of R-SQL

The formal syntax of R-SQL is defined by the grammar in Figure 1. In this grammar, productions start with lowercase letters whereas terminals start with uppercase (SQL terminal symbols

db	::=	R sch := sel_stm; ... R sch := sel_stm;
sch	::=	(A T, ..., A T)
sel_stm	::=	SELECT exp, ..., exp [FROM R, ..., R [WHERE wcond]] | sel_stm UNION sel_stm | sel_stm EXCEPT R
exp	::=	C | R.A | exp opm exp | − exp
wcond	::=	TRUE | FALSE | exp opc exp | NOT(wcond) | wcond [AND | OR] wcond
opm	::=	+ | − | / | ∗
opc	::=	= | <> | < | > | >= | <=

R stands for relation names, A for attribute names, T for standard SQL types and C for constants belonging to a valid SQL type.

Figure 1: A Grammar for the R-SQL Language

use small caps). As usual, optional statements are delimited by square brackets and alternative sentences are separated by pipes.

The language R-SQL overcomes some limitations present in current RDBMS's following SQL:1999. These languages use NOT IN and EXCEPT clauses to deal with negation, and WITH RECURSIVE to engage recursion. As it is pointed out in [GUW09], SQL:1999 does not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause.

A bundle of R-SQL database examples can be found with the system distribution. Next, we present some of them, to show the expressiveness of the definition language. Each of them is intended to illustrate a concrete aspect of the language in a simple and concise way. Section 3 explores a larger and more natural example.

Mutual Recursion Although any mutual recursion can be converted to direct recursion by inlining [KRP93], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance. For instance, the following R-SQL database defines the relations `even` and `odd`, as the classical specification of even and odd numbers up to a bound (100 in the example):

```
even(x float) := SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100;

odd(x float) := SELECT even.x+1 FROM even WHERE even.x<100;
```

Nonlinear Recursion The standard SQL restricts the number of allowed recursive calls to be only one. Here we show how to specify Fibonacci numbers in R-SQL[1]:

[1] The relations `fib1` and `fib2` simply represent two aliases for `fib`, which are necessary because, for simplicity, we have not introduced the usual syntax for renamings in the grammar of Figure 1.

```
fib1(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib2(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib(n float, f float) := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
  WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

Duplicates and Termination Non termination is another problem that arises associated to recursion when coupled with duplicates. For instance, the following standard SQL query (that considers a finite relation `t`) makes current systems either to reject the query or to go into an infinite loop (some systems allow to impose a maximum number of iterations as a simple termination condition, as DB2):

```
WITH RECURSIVE v(a) AS  SELECT * FROM t UNION ALL SELECT * FROM v
  SELECT * FROM v
```

Nevertheless, the fixpoint computation for the corresponding R-SQL relation:

```
v(a float) := SELECT * FROM t UNION SELECT * FROM v;
```

guarantees termination because duplicates are discarded[2] and `v` does not grow unbounded. The very same termination problem also happens in current RDBMS's with the basic transitive closure over graphs including cycles, but not in R-SQL which ensures termination for finite graphs.

2.2 The meaning of an R-SQL database definition

In [ANSS13] we formalized an operational semantics for the language R-SQL based on stratified negation and fixpoint theory, here we summarize the main ideas.

Stratification is based on the definition of a *dependency graph* $DG_{\mathtt{db}}$ for an R-SQL database `db` that is a directed graph whose nodes are the relation names defined in `db`, and the edges, that can be *negatively labelled*, are determined as follows. A relation definition of the form `R sch` := `sel_stm` in `db` produces edges in the graph from every relation name inside `sel_stm` to `R`. Those edges produced by the relation name that is just to the right of an EXCEPT are negatively labelled.

If there are n relations defined in `db`, and we denote by `RN` the set of the relation names defined in `db`, a *stratification* of `db` is a mapping $str : \mathtt{RN} \to \{1,\ldots,n\}$, such that for every two relations $\mathtt{R}_1, \mathtt{R}_2 \in \mathtt{RN}$ it satisfies:

- $str(\mathtt{R}_1) \leq str(\mathtt{R}_2)$, if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$,
- $str(\mathtt{R}_1) < str(\mathtt{R}_2)$ if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$ with at least one negatively labelled edge.

[2] Note that UNION does not require ALL, as current RDBMS's do.

An R-SQL database db is *stratifiable* if there exists a stratification for it. We denote by *numStr* the maximum stratum of the elements of RN.

Intuitively, a relation name preceded by an EXCEPT operator plays the role of a negated predicate (relation) in the deductive database field. A stratification-based solving procedure ensures that when a relation that contains an EXCEPT in its definition is going to be calculated, the meaning of the inner negated relation has been completely evaluated, avoiding nonmonotonicity, as it is widely studied in Datalog [Ull89].

We say that an interpretation I is the relationship between every relation name R and its instance $I(\mathtt{R})$. Interpretations are classified by strata; an interpretation belonging to a stratum i gives meaning to the relations of strata less or equal to i. If I_1, I_2 are two interpretations of stratum i, we say I_1 *is less or equal than* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if the following conditions are satisfied for every $\mathtt{R} \in \mathtt{RN}$:

- $I_1(\mathtt{R}) = I_2(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$, if $str(\mathtt{R}) = i$.

The meaning of every sel_stm w.r.t. an interpretation I can be understood as the set of tuples (in the current instance represented by I) associated to the corresponding equivalent *RA*-expression, denoted by $[\mathtt{sel\_stm}]^I$. This *RA*-expression is defined as follows: [3]

- $[\text{SELECT } \mathtt{exp}_1,\ldots,\mathtt{exp}_k \text{ FROM } \mathtt{R}_1,\ldots,\mathtt{R}_m \text{ WHERE } \mathtt{wcond}]^I = \pi_{\mathtt{exp}_1,\ldots,\mathtt{exp}_k}(\sigma_{\mathtt{wcond}}(I(\mathtt{R}_1)\times\ldots\times I(\mathtt{R}_m)))$
- $[\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]^I = [\mathtt{sel\_stm}_1]^I \cup [\mathtt{sel\_stm}_2]^I$
- $[\mathtt{sel\_stm} \text{ EXCEPT } \mathtt{R}]^I = [\mathtt{sel\_stm}]^I - I(\mathtt{R})$

Example 1 Consider the definitions of the relations odd and even of Section 2. Let us assume a concrete interpretation I such that $I(\mathtt{even}) = \{(0),(2)\}$ and $I(\mathtt{odd}) = \emptyset$. Hence, the interpretation of the select statement that defines the relation odd w.r.t. I is:

$$[\mathtt{SELECT\ even.x+1\ FROM\ even\ WHERE\ even.x<100}]^I = \{(\mathtt{even.x}+1)[a/\mathtt{even.x}] \mid (a) \in I(\mathtt{even}), (\mathtt{even.x}<100)[a/\mathtt{even.x}] \text{ is satisfied } \} = \{(1),(3)\}$$

The case of the relation even is analogous:

$$[\mathtt{SELECT\ 0\ UNION\ SELECT\ odd.x+1\ FROM\ odd\ WHERE\ odd.x<100}]^I = [\mathtt{SELECT\ 0}]^I \cup [\mathtt{SELECT\ odd.x+1\ FROM\ odd\ WHERE\ odd.x<100}]^I = \{(0)\} \cup \{(\mathtt{odd.x}+1)[a/\mathtt{odd.x}] \mid (a) \in I(\mathtt{odd}), (\mathtt{odd.x}<100)[a/\mathtt{odd.x}] \text{ is satisfied}\} = \{(0)\}$$

Notice that the interpretation $\hat{I}$ defined by:

$$\hat{I}(\mathtt{even}) = \{(0),(2),\ldots,(100)\} \text{ and } \hat{I}(\mathtt{odd}) = \{(1),(3),\ldots,(99)\}$$

[3] Notice that arithmetic expressions are allowed as arguments in *projection* (π) and *select* (σ) operations.

satisfies:

$$\hat{I}(\texttt{even}) = [\texttt{SELECT 0 UNION SELECT odd.x + 1 FROM odd WHERE odd.x < 100}]^{\hat{I}}$$
$$\hat{I}(\texttt{odd}) = [\texttt{SELECT even.x + 1 FROM even WHERE even.x < 100}]^{\hat{I}}$$

So, to give meaning to a database definition, we are interested in an interpretation, called fix, such that for every $\texttt{R} \in \texttt{RN}$, if sel_stm is the definition of R, then $fix(\texttt{R}) = [\texttt{sel\_stm}]^{fix}$. In the previous example fix will be $\hat{I}$. Since R can occur inside its definition, for every stratum i, the appropriate interpretation fix_i that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator. These fixpoint interpretations are sequentially constructed from stratum 1 to $numStr$. fix represents the fixpoint of the last stratum and provides the semantics for the whole database.

For every i, $1 \leq i \leq numStr$, we define the continuous operator T_i that transforms interpretations belonging to a stratum i as follows:

- $T_i(I)(\texttt{R}) = I(\texttt{R})$, if $str(\texttt{R}) < i$.
- $T_i(I)(\texttt{R}) = [\texttt{sel\_stm}]^I$, if $str(\texttt{R}) = i$ and R sch := sel_stm is the definition of R in db.
- $T_i(I)(\texttt{R}) = \emptyset$, if $str(\texttt{R}) > i$.

The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$, where $\emptyset(\texttt{R}) = \emptyset$ for every $\texttt{R} \in \texttt{RN}$. We will denote $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$ by fix_1, i.e., fix_1 represents the least fixpoint at stratum 1.

Consider now the sequence $\{T_2^n(fix_1)\}_{n\geq 0}$ of interpretations of stratum 2, greater than fix_1. Using the definition of T_i and the fact that $fix_1(\texttt{R}) = \emptyset$ for every R such that $str(\texttt{R}) \geq 2$, it is easy to prove, by induction on $n \geq 0$, that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2(T_2(fix_1)) \sqsubseteq_2 \ldots \sqsubseteq_2 T_2^n(fix_1), \ldots$$

$\{T_2^n(fix_1)\}_{n\geq 0}$ is a chain that has as least upper bound, $\bigsqcup_{n\geq 0} T_2^n(fix_1)$, which is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2. By proceeding successively in the same way it is possible to find fix_{numStr}. In [ANSS13] we have proved that fix_{numStr} is the interpretation fix we are looking for, that associates the set of tuples denoted by its definition to every relation of the database .

3 The Improved R-SQL System

Here we present the R-SQL system, which is based on the fixpoint construction of the previous section. We describe its structure, focusing on the improvements that increase the efficiency of the previous prototype, presented in [ANSS13]. These enhances are essentially due to the stratification described in Section 3.1 and in the factoring-out process incorporated in the fixpoint algorithm presented in Section 3.2.

As we show in Figure 2, the system is loaded in SWI-Prolog to process an R-SQL database definition. First, the system parses the input database, then it builds the dependency graph and the stratification if it exists (it raises an error, otherwise); finally, it produces a Python script that will create the SQL database in an RDBMS. After this process, the user can connect to

the RDBMS in order to query or modify the database. Although we are referring to PostgreSQL in the concrete implementation https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQLplus, it can be straightforwardly applied to other systems.

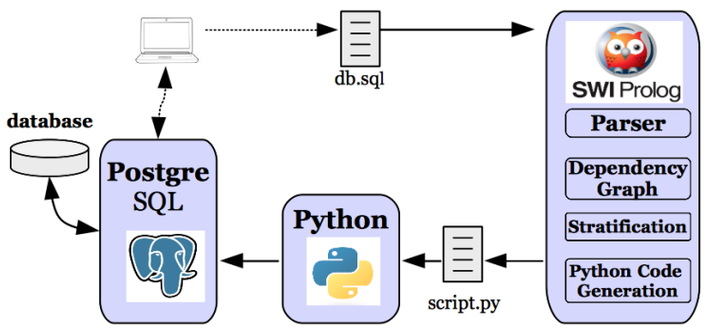


Figure 2: R-SQL System Structure.

Next we present a database for flights to illustrate the process and also will be the working example for the rest of the section. As usual, the information about direct flights can be composed of the city of origin, the city of destination, and the length of the flight. Cities (Lisbon, Madrid, Paris, London, New York) will be represented with constants (`lis`, `mad`, `par`, `lon`, `ny`, resp.). The relation `reachable` consists of all the possible trips between the cities of the database, maybe concatenating more than one flight. The relation `travel` is analogous but also gives time information about alternative trips.

```
flight(frm varchar(10), to varchar(10), time float) :=
  SELECT 'lis','mad',1.0 UNION SELECT 'mad','par',1.5 UNION
  SELECT 'par','lon',2.0 UNION SELECT 'lon','ny',7.0  UNION
  SELECT 'par','ny',8.0;

reachable(frm varchar(10), to varchar(10)) :=
  SELECT flight.frm, flight.to FROM flight UNION
    SELECT reachable.frm, flight.to
    FROM reachable,flight WHERE reachable.to = flight.frm;

travel(frm varchar(10), to varchar(10), time float) :=
  SELECT flight.frm, flight.to, flight.time FROM flight UNION
  SELECT flight.frm, travel.to, flight.time+travel.time
    FROM flight, travel WHERE flight.to = travel.frm;
```

Both `reachable` and `travel` represent transitive closures of the relation `flight`. Notice that if `flight` has a cycle, then the relation `travel` that includes times for each trip is infinite, while `reachable` is not. As pointed before, `reachable` can be finitely computed in our system. But, as `travel` would produce an infinite set of different tuples, some computation limitation would have to be imposed (as the maximum time for a `travel`, for example). However, this is not a drawback of our approach, but an issue due to using infinite relations (built

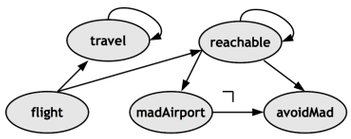


Figure 3: $DG_{\mathtt{db}}$ of the working example.

with arithmetic expressions). The relation `madAirport` contains travels departing or arriving in Madrid, while `avoidMad` contains the possible travels that neither begin, nor end in Madrid.

```
madAirport(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  WHERE (reachable.frm = 'mad' OR reachable.to = 'mad');

avoidMad(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable EXCEPT madAirport;
```

This definition includes negation together with recursive relations. This combination can not be expressed in SQL:1999 as it is shown in [FMMP96]. The dependency graph of this database is depicted in Figure 3, where negatively labelled edges are annotated with ¬.

3.1 Stratification

Given a database and its dependency graph, there can be a number of different stratifications for it. For instance, for the dependency graph of Figure 4 a possible stratification can assign stratum 1 to the relations $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\}$ and stratum 2 to $\{\mathtt{f},\mathtt{g}\}$.

For the graph of Figure 4, intuitively it is easy to see that only `b` and `c` must belong to the same stratum due to the mutual dependency between them. The next algorithm minimizes the number of relations in each stratum, which allows to enhance the efficiency of the fixpoint computation as shown in Section 3.2.

- Compute the *strongly connected components* C from DG_{db}. Negative labels are not relevant initially, but once the components are evaluated, it must be checked if there exists some cycle with a negatively labeled edge. In such a case, `db` is not stratifiable and the computation stops. For the example of Figure 4 the components are $\{\mathtt{a}\}$, $\{\mathtt{f}\}$, $\{\mathtt{g}\}$, $\{\mathtt{b},\mathtt{c}\}$, $\{\mathtt{d}\}$ and $\{\mathtt{e}\}$.

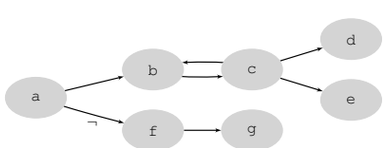


Figure 4: Dependency Graph Example

- *Collapse each strongly connected component* obtaining a new graph with a node for each component, C, and with an edge from C to C' if and only if C contains a relation R and C' contains a relation R′, such that there is an edge from R to R′ in DG_{db}. In our example, the component $\{\mathtt{b},\mathtt{c}\}$ can be collapsed to the node bc, and the rest to its single element. The new graph has the edges $\{\mathtt{a}\to\mathtt{bc},\mathtt{bc}\to\mathtt{d},\mathtt{bc}\to\mathtt{e},\mathtt{a}\to\mathtt{f},\mathtt{f}\to\mathtt{g}\}$.

- Obtain a *topological sorting* for the resulting graph. In our example we can get the sorting $\mathtt{a}<\mathtt{f}<\mathtt{g}<\mathtt{bc}<\mathtt{e}<\mathtt{d}$.

- Uncollapse the nodes of such a sorting for obtaining a topological sorting for the strongly connected components, and enumerate them in ascending order. In our example, we get $\{\mathtt{a}\}<\{\mathtt{f}\}<\{\mathtt{g}\}<\{\mathtt{b},\mathtt{c}\}<\{\mathtt{e}\}<\{\mathtt{d}\}$.

  Then, the expected stratification $str(\mathtt{a})=1$; $str(\mathtt{f})=2$; $str(\mathtt{g})=3$; $str(\mathtt{b})=str(\mathtt{c})=4$; $str(\mathtt{e})=5$; $str(\mathtt{d})=6$ is obtained.

The concrete implementation of this algorithm in R-SQL uses the library *ugraphs* of SWI-Prolog and the module *scc* implemented by Markus Triska, accessible from http://www.logic.at/prolog/scc.pl. For the dependency graph of Figure 3, R-SQL assigns stratum 1 to flight, 2 to travel, 3 to reachable, 4 to madAirport, and 5 to avoidMad.

3.2 The Computation of the Database Fixpoint

Next, we present the algorithm for generating the SQL database corresponding to the fixpoint of an R-SQL database definition db. This algorithm is shown in Figure 5. It produces the SQL statements (CREATE and INSERT) needed to build such a database.

```
 1  for all R ∈ RN_db do
 2    CREATE TABLE R sch;
 3  end for
 4  i := 1
 5  while i ≤ numStr do
 6    for all R ∈ RN_i do
 7      INSERT INTO R out(sel_stm_R);
 8    end for
 9    repeat
10      size := rel_size(RN_i)
11      for all R ∈ RN_i do
12        INSERT INTO R in(sel_stm_R) EXCEPT SELECT * FROM R;
13      end for
14    until size = rel_size(RN_i)
15    i := i + 1
16  end while
```

Figure 5: Algorithm to Compute the Fixpoint

The algorithm considers a concrete stratification for the database where *numStr* denotes the number of strata and $\mathtt{NR}_i$ the set of relations of stratum i. First of all, a table is created for each relation `R sch :=` $\mathtt{sel\_stm}_\mathtt{R}$ of the database (lines 1-3). Then, the external *while* at line 5 computes successively the fixpoints $fix_1, fix_2, \ldots, fix_{numStr}$. Following the semantics, each fix_i is calculated for every relation of $\mathtt{NR}_i$, by iterating the fixpoint operators T_i, i.e., the internal *repeat* (lines 9-14) at iteration n computes $T_i^n(fix_{i-1})$. The loop is iterated while some tuple is added to the tables of the current stratum; the variable *size* is used to check this condition.

This algorithm enhances the introduced in [ANSS13] by reducing the work in the iterations of the *repeat*, i.e., simplifying the operations done for filling the tables, so improving the efficiency. The idea is that the iteration of the operator T_i is only needed for recursive relations, and even more precisely, only for the recursive fragment of the select statements defining those relations. With this aim we have defined the functions *in* and *out* to split each `sel_stm` into, respectively, the (recursive) fragment that must be used in the INSERT statements inside the loop, and the fragment that can be processed before the loop, as the base case of the recursive definition. Then, the *for* at lines 6-8 processes the *out* fragments, and the INSERT's at lines 11-13 only process the *in* fragments. The *in* and *out* fragments of a `sel_stm` can be easily determined using the stratum of its components because the stratification defined in the Section 3.1 is such that if a relation `R` in stratum i depends on another relation $\mathtt{R}'$, then the stratum of $\mathtt{R}'$ is lower than i, so it must be previously computed, or it is exactly i (if they are mutually recursive) and both relations must be computed simultaneously. Therefore, if for instance `R :=` $\mathtt{sel\_stm}_1$ UNION $\mathtt{sel\_stm}_2$, $str(\mathtt{R}) = i$, and $str(\mathtt{sel\_stm}_1) < i$, then $\mathtt{sel\_stm}_1$ will be part of the *out* fragment, and the corresponding tuples can be inserted before the loop, because the involved relations are already computed in the computation of a previous stratum. Functions *in* and *out* can be easily defined using the stratification as follows:

If $str(\mathtt{sel\_stm}) < i$ then we have:

- $in(\mathtt{sel\_stm}) = \emptyset$ and $out(\mathtt{sel\_stm}) = \mathtt{sel\_stm}$.

If $str(\mathtt{sel\_stm}) = i$ then, the functions are defined by recursion on the structure of `sel_stm`:

- `sel_stm` ≡ SELECT `exp` ... `exp` FROM `R` ... `R` WHERE `wcond`
  $in(\mathtt{sel\_stm}) = \mathtt{sel\_stm}$ and $out(\mathtt{sel\_stm}) = \emptyset$
- $\mathtt{sel\_stm} \equiv \mathtt{sel\_stm}_1$ UNION $\mathtt{sel\_stm}_2$
  - If $str(\mathtt{sel\_stm}_1) = str(\mathtt{sel\_stm}_2) = i$ then:
    $in(\mathtt{sel\_stm}) = in(\mathtt{sel\_stm}_1)$ UNION $in(\mathtt{sel\_stm}_2)$ and
    $out(\mathtt{sel\_stm}) = out(\mathtt{sel\_stm}_1)$ UNION $out(\mathtt{sel\_stm}_2)$
  - If $str(\mathtt{sel\_stm}_1) = i$ and $str(\mathtt{sel\_stm}_2) < i$ then:
    $in(\mathtt{sel\_stm}) = in(\mathtt{sel\_stm}_1)$ and $out(\mathtt{sel\_stm}) = out(\mathtt{sel\_stm}_1)$ UNION $\mathtt{sel\_stm}_2$
  - If $str(\mathtt{sel\_stm}_1) < i$ and $str(\mathtt{sel\_stm}_2) = i$ then:
    $in(\mathtt{sel\_stm}) = in(\mathtt{sel\_stm}_2)$ and $out(\mathtt{sel\_stm})$ = $\mathtt{sel\_stm}_1$ UNION $out(\mathtt{sel\_stm}_2)$
- $\mathtt{sel\_stm} \equiv \mathtt{sel\_stm}_1$ EXCEPT $\mathtt{sel\_stm}_2$
  $in(\mathtt{sel\_stm}) = in(\mathtt{sel\_stm}_1)$ EXCEPT $\mathtt{sel\_stm}_2$ and
  $out(\mathtt{sel\_stm}) = out(\mathtt{sel\_stm}_1)$ EXCEPT $\mathtt{sel\_stm}_2$

The concrete implementation of the algorithm of Figure 5 can be done in a number of ways. We have chosen Python as the host language mainly because it is multiplatform and provides easy connections with different database systems such as PostgreSQL, DB2, MySQL, or even via ODBC, which allows connectivity to almost any RDBMS. The additional features required for the host language are basic: Loops, assignment and simple arithmetic.

Below, we show the Python code generated for the working example of flights. It uses the Python library *psycopg2* (available at `http://initd.org/psycopg/`) which allows to connect to an RDBMS and then submit SQL queries as:

```
cursor.execute("<query>")
```

where `<query>` is any valid SQL query. The generated code expands all the loops of the algorithm of Figure 5, except the *repeat* at lines 9-14. As Python does not provide a *repeat* (or *do-while*) loop construction, we implement it as a *while True* sentence with the corresponding *break* for stopping it when the condition holds. We will show it in the code generated for stratum 2. Moreover, we also implement a Python function `relSize(<list of relations>)` that returns the number of tuples of the relations specified in its argument.

The *for* at lines 1-3 is expanded as:

```
cursor.execute("CREATE table flight
                    (frm varchar(10), to varchar(10), time float);")
cursor.execute("CREATE table travel
                    (frm varchar(10), to varchar(10), time float);")
```

and so on for the rest of relations. Now, we detail some parts of the code generated stratum by stratum. For stratum 1 the *in* fragment is empty and we have:

```
# Code generated for Stratum 1
cursor.execute("INSERT INTO flight
           (SELECT 'lis','mad',1 UNION SELECT 'mad','par',1.5 UNION
            SELECT 'par','lon',2 UNION SELECT 'lon','ny',7 UNION
            SELECT 'par','ny',8) EXCEPT
           SELECT * FROM flight;")
```

Stratum 2 contains the relation `travel` whose definition can be splitted into two parts with the functions *in* and *out*.

```
# Code generated for Stratum 2
# out fragment
cursor.execute("INSERT INTO travel (SELECT * FROM flight);")

# in fragment
while True:
  cursor.execute("INSERT INTO travel
             (SELECT flight.frm,travel.to,flight.time+travel.time
              FROM flight,travel WHERE flight.to = travel.frm)
              EXCEPT SELECT * FROM travel;")

  newSize = relSize(["travel"])

  if (newSize != size):
       size = newSize
  else:
       break
```

The tuples added for `travel` at each iteration of this code are shown in the next Table:

	Set of added tuples
out fragment	{(lon, ny, 7.0), (par, lon, 2.0), (par, ny, 8.0), (mad, par, 1.5), (lis, mad, 1.0)}
in fragment: iteration 1	{(lis, par, 2.5), (par, ny, 9.0), (mad, ny, 9.5), (mad, lon, 3.5)}
in fragment: iteration 2	{(lis, ny, 10.5), (lis, lon, 4.5), (mad, ny, 10.5)}
in fragment: iteration 3	{(lis, lon, 4.5), (mad, ny, 10.5), (lis, ny, 11.5)}

Analogously, the system produces the Python code for strata 3 and 4, which correspond to `reachable` and `madAirport`, respectively. Finally, in the last stratum the `avoidMad` relation is computed (there is no *in* fragment in this case):

```
# Code generated for Stratum 5
# out fragment
cursor.execute("INSERT INTO avoidMad
            (SELECT travel.frm,travel.to FROM travel
             EXCEPT SELECT * FROM madAirport)");
```

This completes the fixpoint for the working example database. The values for `flight`, `madAirport` and `avoidMad` tables are illustrated in the graph in Figure 6. Direct flights are represented in blue color and labeled with their corresponding time. Paths for `madAirport` relation are represented in red color and path for `avoidMad` relation are represented in black color.

Once the R-SQL database definition has been processed, the tables obtained are available as a database instance in PostgreSQL. Then, the user can formulate queries that will be solved using those tables (without performing any further fixpoint computation).

3.3 Performance

This section analyzes the system performance. First, we focus on the improvement of factoring out SQL fragments (as already explained in Section 3.2). And, second, we develop a field analysis by targeting the system to different current state-of-the art relational systems, introducing the

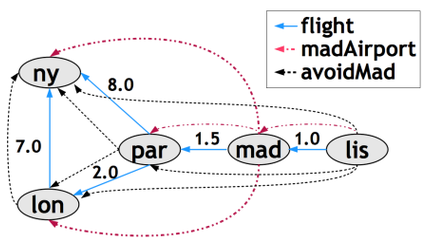


Figure 6: Graphical representation of resulting values of the working example.

benefits of a semi-naïve differential optimization [Ull85] for linear recursive queries. Numbers for tables in this section are expressed in milliseconds and represent the average of a number of runs, where the maximum and minimum have been elided.

3.3.1 Factoring-Out Improvement

As introduced, any DBMS allowing Python access can be used to implement our proposal. This section develops the connection to IBM DB2 as a target system for analyzing the performance. We consider the benchmark `reachable` that implements the transitive closure of the relation `flight`, as introduced in Section 3. To build a parametric relation, we consider links in flights as the tuples $\{(1,2),(2,3),\ldots,(n,n+1)\}$, where $n+1$ is the number of nodes in the graph and the type of the fields have been changed to integer. Table 1 shows the results for instances of this benchmark with a number of tuples ranging from 100 to 350 (first column). Second column lists the number of tuples generated in the result set. Third and fourth columns show the elapsed running time for solving the query in R-SQL with no factoring-out improvement (No FOI) and with this improvement enabled (With FOI), respectively. Fifth column (Speed-up) shows the speed-up due to FOI as a percentage. The last column (Difference) shows the absolute time difference between both timings. Benchmarks have been run on an Intel Core2 Quad CPU at 2.4GHz and 3GB RAM, running Windows XP 32bit SP3, and IBM DB2 Express Edition 10.1.0 database server with a default configuration.

Tuples	Result Tuples	No FOI	With FOI	Speed-up	Difference
100	5,050	1,135	1,050	8.1%	85
150	11,325	4,438	3,428	29.4%	1,010
200	20,100	10,048	8,172	23.0%	1,876
250	31,375	19,001	16,041	18.5%	2,960
300	45,150	32,710	28,381	15.3%	4,329
350	61,425	50,085	44,175	13.4%	5,910

Table 1: Factoring-Out Improvement (FOI)

From this experiment we confirm the expected results for factoring the fragment `select * from flight` out of the recursive clause and the *repeat* loop. Indeed, even for a single SQL fragment as this, speed-ups of up to almost 30% are reached. However, as long as the tuples do increase in the instances, the speed-up decrease because the main computation effort corresponds to the *repeat* loop because of the EXCEPT operator.

Next section deals with other optimizations and comparison with other relational and deductive systems.

3.3.2 Analysis of Systems

This section considers different current state-of-the-art relational systems which include recursive queries: PostgreSQL 9.3, Oracle 11g, and DB2 10.1, all of them with a default configuration. We compare R-SQL when solving the previous benchmark with these systems and show the

importance of introducing the semi-naïve differential optimization [Ull85]. To make the comparison fairest with the RDBMS's, which do not discard duplicates, we omit the operator EXCEPT to behave similarly to the optimized R-SQL systems. Also, we include the last published version of DLV^{DB} in this comparison as a deductive system which is able to project its solving to these external databases when computing a transitive closure. Another related deductive system is LDL++, but unfortunately it is not included in this comparison since it has been replaced by the system DeALS whose binaries and/or sources are not available yet.

RDBMS	System	100	200	300	400	500
PostgreSQL	*Native SQL*	161	187	240	360	713
	R-SQL	500	3,198	12,406	39,802	71,922
	Diff-R-SQL	208	459	1,073	2,271	4,115
	TDiff-R-SQL	260	578	1,323	2,745	5,693
	DLV^{DB}	703	1,651	4,458	8,047	13,120
Oracle	*Native SQL*	604	1,781	5,765	13,349	26,297
	R-SQL	880	3,802	12,057	27,989	56,641
	Diff-R-SQL	708	1,437	3,224	6,240	11,469
	TDiff-R-SQL	646	995	1,708	2,453	3,422
	DLV^{DB}	6,875	12,849	18,912	30,583	42,146
DB2	*Native SQL*	677	1,016	1,323	2,052	3,099
	R-SQL	1,271	5,797	97,052	129,917	150,104
	Diff-R-SQL	698	932	2,672	2,859	3,213
	TDiff-R-SQL	646	1,000	1,578	4,021	9,021
	DLV^{DB}	6,339	12,666	53,552	57,349	100,391

Table 2: Analysis of Systems

The results for different instances of the benchmark are given in Table 2. Numbers are now arranged with the parameter n ranging in the horizontal axis, and rows include the considered RDBMS (first column), the system connected to this relational database (second column), and then (in the next five columns), the wall time for solving each instance (from 100 up to 500 tuples in the relation `flight`, which delivers from 5,050 up to 125,250 tuples in the result set of the query benchmark). Below the headings, lines are arranged in major rows, each one referring to a concrete RDBMS. And, for each RDBMS (*PostgreSQL*, *Oracle*, *DB2*)[4], five minor rows are listed, which refer to each system. The first minor row *Native SQL* refers to the corresponding RDBMS, which is used to compare how the rest of the systems behave w.r.t. a native execution of the benchmark, i.e., resorting to the recursive query specification for the transitive closure that each RDBMS provides. For instance, DB2 uses the following syntax (where `rec` is the temporary recursive relation which is built to fill the relation `reachable`):

```
INSERT INTO reachable
  WITH rec(frm,to) AS
```

[4] Incidentally, MySQL does not support recursive queries at all.

```
    (SELECT * FROM flight
     UNION ALL
     SELECT flight.frm, rec.to FROM flight,rec
     WHERE flight.to = rec.frm)
   SELECT * FROM rec;
```

The next minor row *R-SQL* refers to the implementation we have presented in Section 3. Minor row labeled with *Diff-R-SQL* presents the results for R-SQL with the semi-naïve differential optimization enabled as explained in [Ull85]. Roughly, for a linear query, this optimization refers to use in each iteration only the results that have been generated in the previous iteration to build new tuples. To implement this, we have resorted to add a new integer column (`it` in the benchmark) holding the iteration in which a given tuple has been generated. For instance, the next query is executed for each iteration `$IT$` (this is substituted by the actual iteration number along iterations):

```
INSERT INTO reachable
  SELECT flight.frm, reachable.to, $IT$
  FROM flight, reachable
  WHERE flight.to = reachable.frm AND
        reachable.it = $IT$-1;
```

Next, the row labeled with *TDiff-R-SQL* refers to an alternative implementation of the semi-naïve differential optimization, which consists on storing all the tuples generated in a given iteration in a temporary table. Then, the join at each iteration is computed between `flight` and this temporary table, therefore avoiding to scan the growing relation `reachable` looking for the tuples with a given iteration number value in the extra field. In fact, two temporary tables are needed: One for accessing the tuples generated in the previous iteration, and another one to store the new tuples. Next, there is a sketch of the SQL statements submitted in each iteration to DB2, where `reachable_temp1` is intended to hold the tuples generated in the previous iteration, and `reachable_temp2` is for the current one (temporary tables are preceded by `SESSION.`):

```
INSERT INTO SESSION.reachable_temp2
  SELECT flight.ori, SESSION.reachable_temp1.des
  FROM flight, SESSION.reachable_temp1
  WHERE flight.des = SESSION.reachable_temp1.ori;
...
INSERT INTO reachable SELECT * FROM SESSION.reachable_temp1;
DELETE FROM SESSION.reachable_temp1;
INSERT INTO SESSION.reachable_temp1
  SELECT * FROM SESSION.reachable_temp2;
DELETE FROM SESSION.reachable_temp2;
```

The first SQL sentence loads into `reachable_temp2` the results just computed for the current iteration. Next sentences simply load on `reachable` the results from the previous iteration, and transfer the results just available in `reachable_temp2` to `reachable_temp1` in order for them to be available for the next iteration. `reachable_temp2` is finally flushed to be ready for the next iteration as well.

Using temporary tables should reveal an advantage as neither log records nor lock management are needed. They are computed in-memory as much as possible; only when they do not fit into RAM, memory space quota is requested for them.

Finally, the row labeled with DLV^{DB} stands for this deductive system, which uses the same ODBC bridge to access those relational systems.

Looking at the numbers, it is noticeable that the best performance is achieved by the native SQL execution in PostgreSQL for all the considered instances ($n \in \{100, 200, \ldots, 500\}$) of the benchmark. Also, the worst performance corresponds to R-SQL without optimizations (and including the operator EXCEPT), which is also clear as the join and the difference must be processed in each iteration for all the tuples, including those that definitely will not be involved in generating a new one. The semi-naïve differential optimization (which also avoids the operator EXCEPT) alleviates this enormously, with a huge factor of $150,104/3,213 = 46.7\times$, when comparing *R-SQL* vs. *Diff-R-SQL* for DB2. DLV^{DB} is the next system in the performance ranking, behaving better than R-SQL but worse than the rest. Depending on the RDBMS, the next best system can be either *Diff-R-SQL* or *TDiff-R-SQL*: The first one performs better than the second for PostgreSQL and the other way round for Oracle and DB2. Noticeably, both perform better than *Native SQL* for Oracle, and *Diff-R-SQL* behaves roughly similar to DB2. These numbers highlight how similar techniques are differently managed by the different RDBMS's. For example, whereas for Oracle the use of temporary tables is of paramount importance for lowering the solving time, its effect is the contrary for DB2. We have also tested table functions, which provide a way to implement parametric views. However, they do not provide better performance than the already illustrated optimizations (and Oracle faces the mutating table problem when using them to insert tuples in the same source table).

All in all, in the best case we are able to beat an RDBMS by a factor of $26,297/3,422 = 7.7\times$, and in the worst case (but considering the best optimization) we are beaten by a factor of $4,115/713 = 5.8\times$. To better understand this slowdown, we must consider that the R-SQL system runs an interpreted script (Python) and in each iteration, one or several SQL statements are sent to the RDBMS via the ODBC bridge. SQL statements sent in this way must be compiled by the RDBMS for each iteration, so that it becomes a significant burden on the system, together with the communication cost due to the bridge. Therefore, using a compiled language supporting prepared SQL statements should be a point worth to explore for performance gains.

4 Conclusions

R-SQL has been designed to compute the meaning of a database definition and then to query this database. Notice that the modification of a relation of the database in the underlying RDBMS can cause inconsistencies since the tables are not recomputed. For instance, after processing the database for flights, if the user adds or deletes a tuple for the relation `flight`, then the relation `travel` will become inconsistent according to its R-SQL definition. But this is the very same behavior of RDBMS's when dealing with materialized views. A future direction in order to fully integrate R-SQL into an RDBMS is to have the possibility of restoring the consistence of the database (using triggers for instance), as well as to define additional (possibly recursive) views. This restoring involves the recomputation of the database fixpoint. But, using the dependency graph, it is easy to determine the subset of relations that must be calculated, instead of computing the whole fixpoint for the database. Moreover, those relations may not need to be recomputed from scratch. In addition, it is straightforward to modify the algorithm introduced in Section

3.2 to get a lazy evaluation of such relations, performing iterations only when new values are demanded.

As shown in Section 3.3.2, the semi-naïve differential optimization [Ull89] for linear recursive queries has a notable impact on performance. Nonetheless, our system can be further extended for non linear recursive queries and with enhancements as in [ZCF+97, BR87], as DLV [TLLP08] does. Implementing all these optimizations are left for future work.

Although our proposal is encouraging as results reveal, efficiency can also be improved by indexing (e.g., tries [SW12] and BDD's [WACL05]) temporary relations during fixpoint computations. To seamlessly integrate this into an RDBMS, we can profit from the fourth-generation languages (e.g., SQL PL in IBM DB2 and PL/SQL in Oracle) and completely integrate query solving and view maintenance into the RDBMS. This way, prepared SQL statements are available in a compiled setting, which should also improve performance. We are currently extending the R-SQL system with the enhancements aforementioned and more features as hypothetical definitions and aggregates.

Bibliography

[ANSS13] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández. Formalizing a Broader Recursion Coverage in SQL. In *Symposium on Practical Aspects of Declarative Languages (PADL'13)*. LNCS 7752, pp. 93 – 108. 2013.

[AOT+03] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo. The Deductive Database System LDL++. *TPLP* 3(1):61–94, 2003.

[BR87] I. Balbin, K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *J. Log. Program.* 4(3):259–262, 1987.

[Cod70] E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM* 13(6):377–390, June 1970.

[Dat09] C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.

[FMMP96] S. J. Finkelstein, N. Mattos, I. S. Mumick, H. Pirahesh. Expressing Recursive Queries in SQL. Technical report, ISO, 1996.

[GUW09] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[KRP93] O. Kaser, C. R. Ramakrishnan, S. Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.* 2(1-4):151–164, Mar. 1993.

[MP94] I. S. Mumick, H. Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Rec.* 23:103–114, May 1994.

[SP13] F. Sáenz-Pérez. Towards Bridging the Expressiveness Gap Between Relational and Deductive Databases. In *XIII Jornadas sobre Programación y Lenguajes, PROLE2013 (SISTEDES)*. September 2013.

[SW12] T. Swift, D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP* 12(1-2):157–187, 2012.

[TLLP08] G. Terracina, N. Leone, V. Lio, C. Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP* 8(2):129–165, 2008.

[Ull85] J. D. Ullman. Implementation of Logical Query Languages for Databases. *ACM Trans. Database Syst.* 10(3):289–321, 1985.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1989.

[WACL05] J. Whaley, D. Avots, M. Carbin, M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *In Proceedings of Programming Languages and Systems: Third Asian Symposium*. 2005.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., 1997.